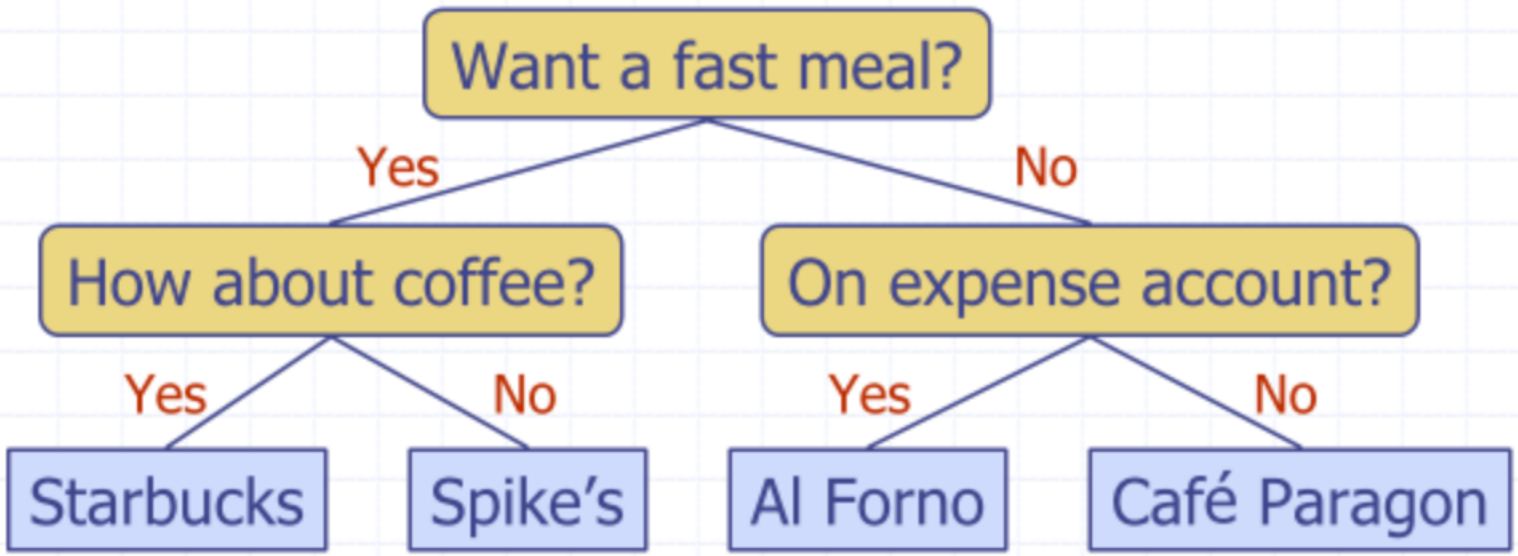
Decision Tree

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



Properties of binary trees:

h = height

e = # of external nodes = # leaves

i = # of internal nodes

n = total # of nodes

(a) Show a relation between e & i
(Hint: You could use recursion)

(b) Show an upperbound & lowerbound for h in terms of n

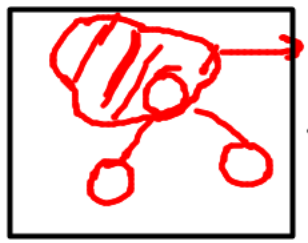
If your finding is only for "proper" binary trees, please state so.

Solutions / suggestions

(a) (i) $e = i + 1$ for proper tree : Proof by induction

Base: $i = 1$

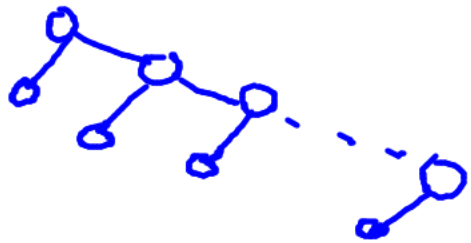
Assume for $i \le k$. Let $i = k+1$



has $i \& e$

→ has $i+1$ & $e+2-1$
$= e+1$

(b) (i) $h \le \left\lceil \frac{n}{2} \right\rceil$



for proper

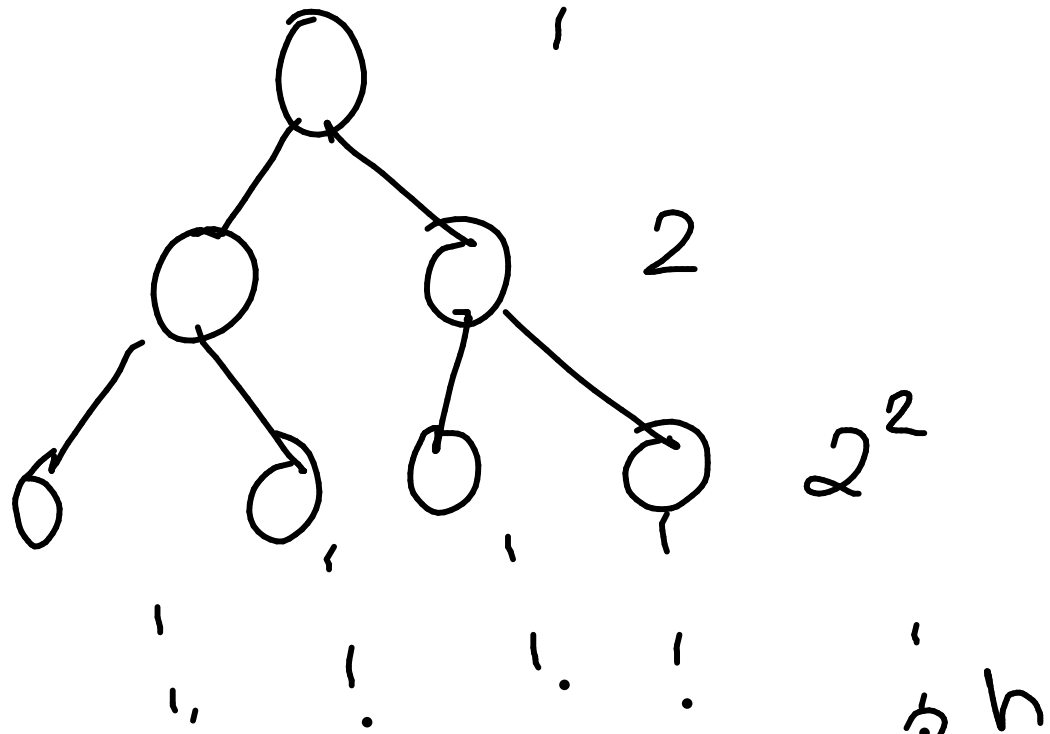
(ii) $h \ge \log_2(n+1)$

(v) $h \ge \log_2(n+1) - 1$

(iii) $h \le \left\lceil \frac{n-1}{2} \right\rceil$

(iv) $h \ge \log_2 n + 1$

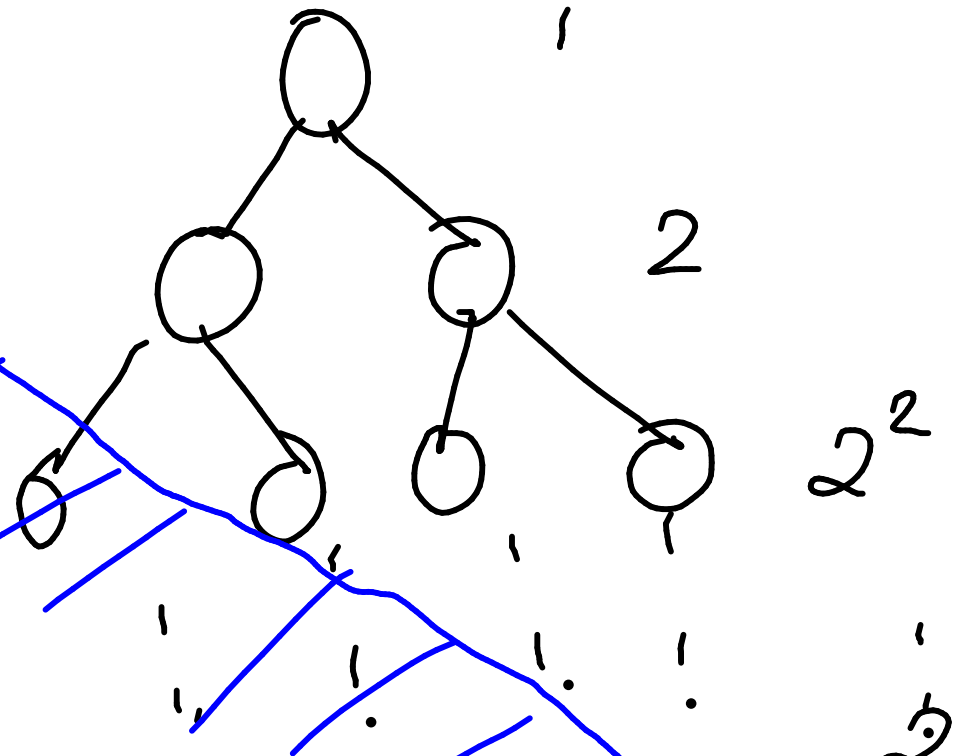
One more inequality for arbitrary binary tree by considering balanced binary trees



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

$(h+1)$

One more inequality for arbitrary binary tree by considering balanced binary trees

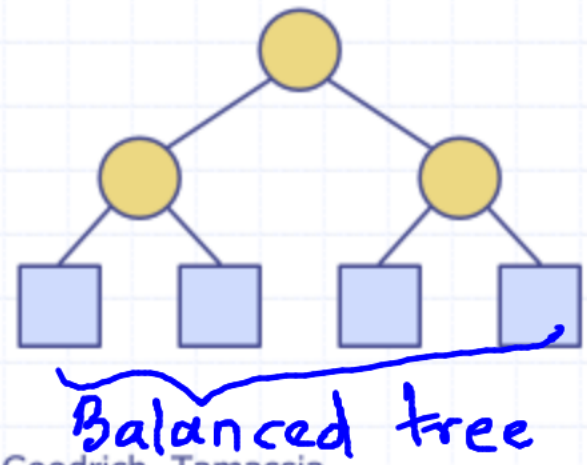


In general for non-balanced tree, n can only be less

$$n \leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1 \Rightarrow \boxed{h \geqslant \log_2^{(h+1)}(n+1) - 1}$$

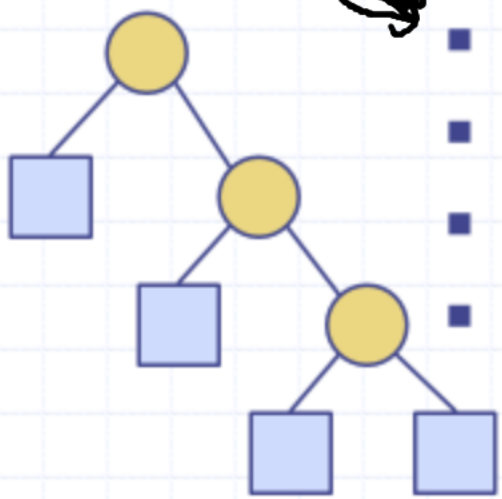
Properties of Proper Binary Trees

- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height



Balanced tree

- Properties:
  - $e = i + 1$ → $n = i + e$
  - $n = 2e - 1$ ($n = 2i + 1$)
  - $h \leq i$ → since all ancestors are internal nodes
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$ [$e = 2^h$ for balanced tree]
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$



Not necessarily

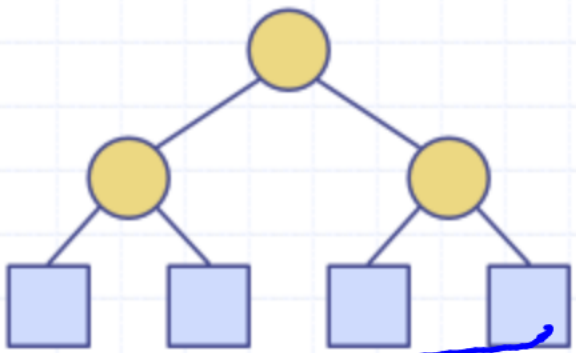
Properties of Proper Binary Trees

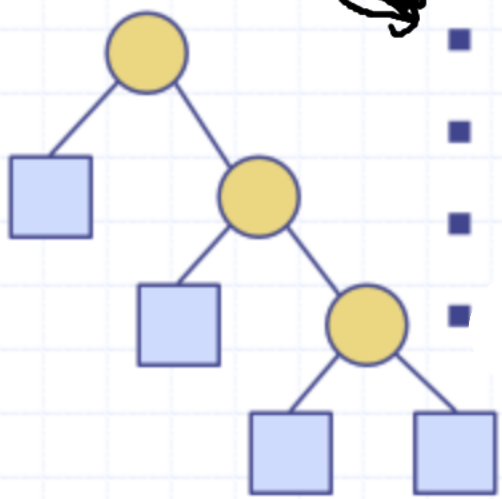
- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height

- Properties:
  - $e \leq i + 1$
  - $n \geq 2e - 1$
  - $h \leq i$ → since all ancestors are internal nodes
  - ~~$h \leq (n - 1)/2$~~
  - $e \leq 2^h$ [$e = 2^h$ for balanced tree]
  - $h \geq \log_2 e$
  - $h \geq \log_2(n+1) - 1$

$n = i + e$

$n \leq 2i + 1$

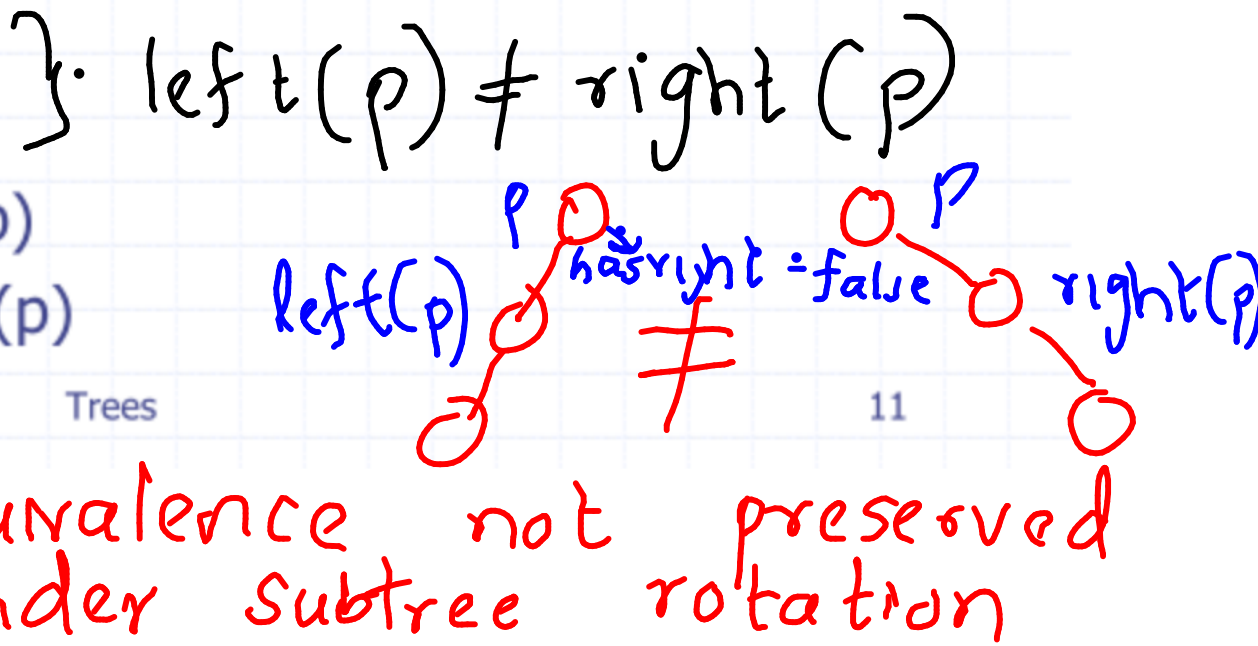




Balanced tree

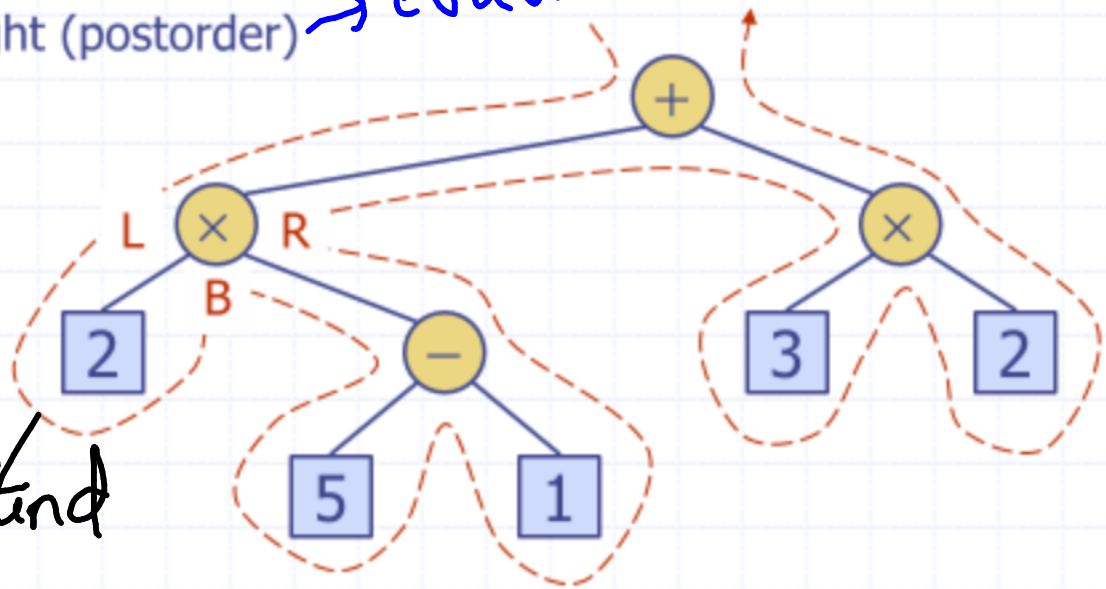
BinaryTree ADT (§ 6.3.1)

- 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)
  - boolean hasLeft(p)
  - boolean hasRight(p)
- Update methods may be defined by data structures implementing the BinaryTree ADT



Euler Tour Traversal : Handy to write single program with flags for 3 cases

- Generic traversal of a binary tree
- Includes a special cases the preorder, postorder and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (preorder)
  - from below (inorder) → print arithmatic expr
  - on the right (postorder) → evaluate arithmatic expr



Blanket around rec:

PRE: Process node at L
POST: Process node at R
IN: Process node at B

Template Method Pattern

- Generic algorithm that can be specialized by redefining certain steps
- Implemented by means of an abstract Java class
- Visit methods that can be redefined by subclasses
- Template method eulerTour
  - Recursively called on the left and right children
  - A Result object with fields leftResult, rightResult and finalResult keeps track of the output of the recursive calls to eulerTour

```
public abstract class EulerTour {
    protected BinaryTree tree;
    protected void visitExternal(Position p, Result r) { }
    protected void visitLeft(Position p, Result r) { }
    protected void visitBelow(Position p, Result r) { }
    protected void visitRight(Position p, Result r) { }
    protected Object eulerTour(Position p) {
        Result r = new Result();
        if tree.isExternal(p) { visitExternal(p, r); }
            else {
                visitLeft(p, r);
                r.leftResult = eulerTour(tree.left(p));
                visitBelow(p, r);
                r.rightResult = eulerTour(tree.right(p));
                visitRight(p, r);
                return r.finalResult;
            } …
```

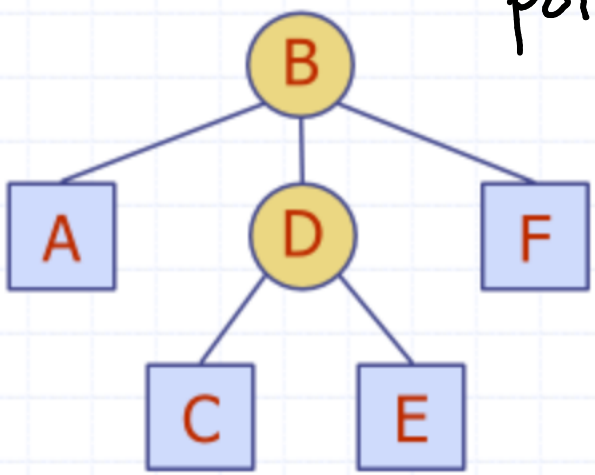
Specializations of EulerTour

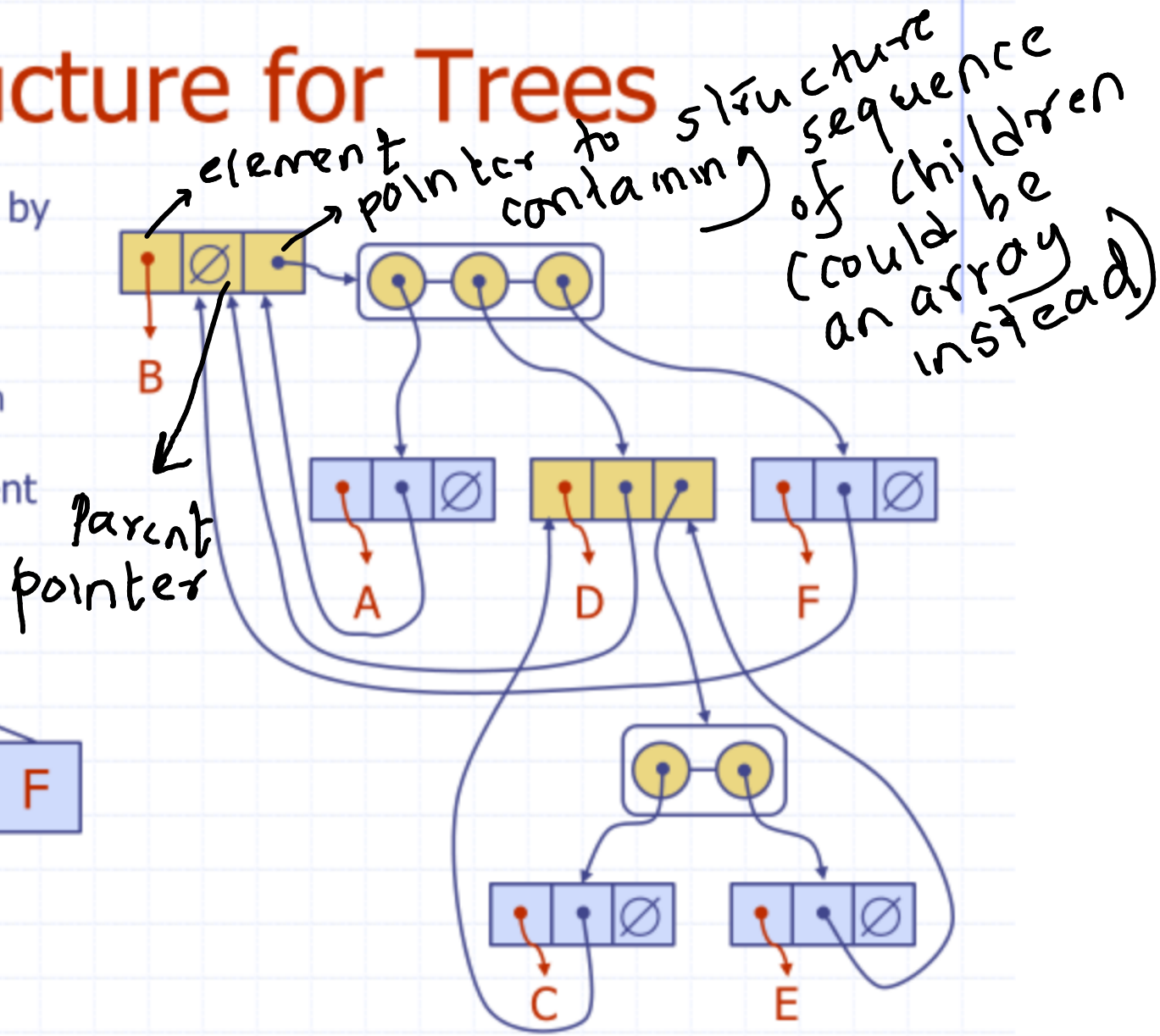
- We show how to specialize class EulerTour to evaluate an arithmetic expression
- Assumptions
  - External nodes store Integer objects
  - Internal nodes store Operator objects supporting method operation (Integer, Integer)

```
public class EvaluateExpression
                    extends EulerTour {

  protected void visitExternal(Position p, Result r) {
    r.finalResult = (Integer) p.element();
  }

  protected void visitRight(Position p, Result r) {
    Operator op = (Operator) p.element();
    r.finalResult = op.operation(
                        (Integer) r.leftResult,
                        (Integer) r.rightResult
                        );
  }

  ...
}
```

Linked Structure for Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes
- Node objects implement the Position ADT





Points based on linked structure representation for tree:

1. Can we use a "vector" or fixed length "array" for the list of children?
2. Does storing parent pointer help in efficient tree traversal?

[see next page]

3. How about vector/array to represent entire tree

Simplest case: Balanced binary tree:

Example:

Evaluate Arithmetic Expressions

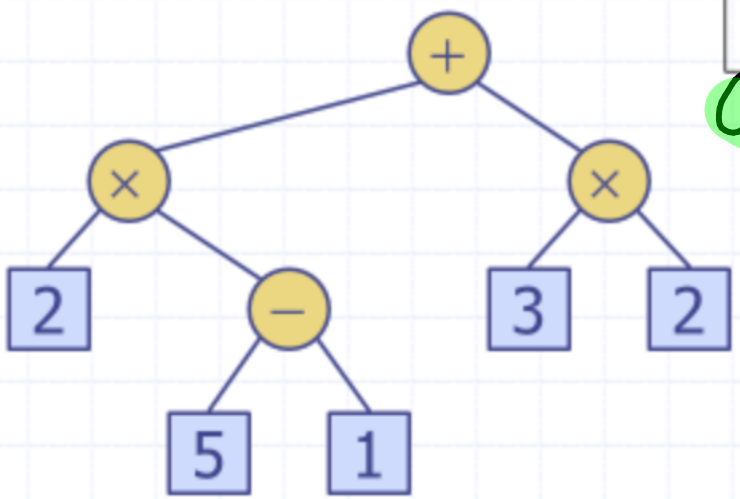
- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

```
Algorithm evalExpr(v)
    if isExternal (v)
        return v.element ()
    else
        x ← evalExpr(leftChild (v))
        y ← evalExpr(rightChild (v))
        ◊ ← operator stored at v
        return x ◊ y
```

using child pointer

Postorder

(Does not need parent pointers)

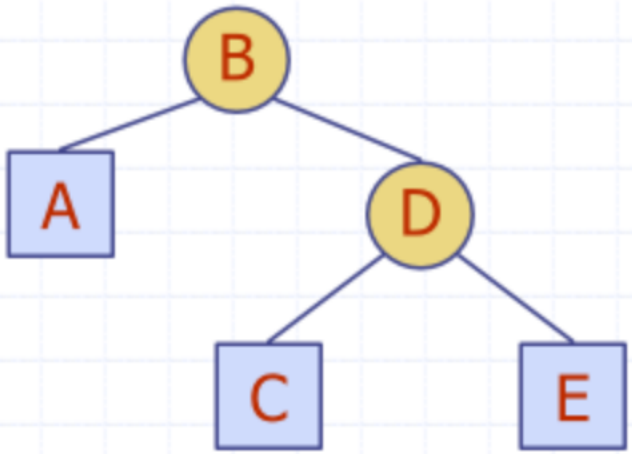


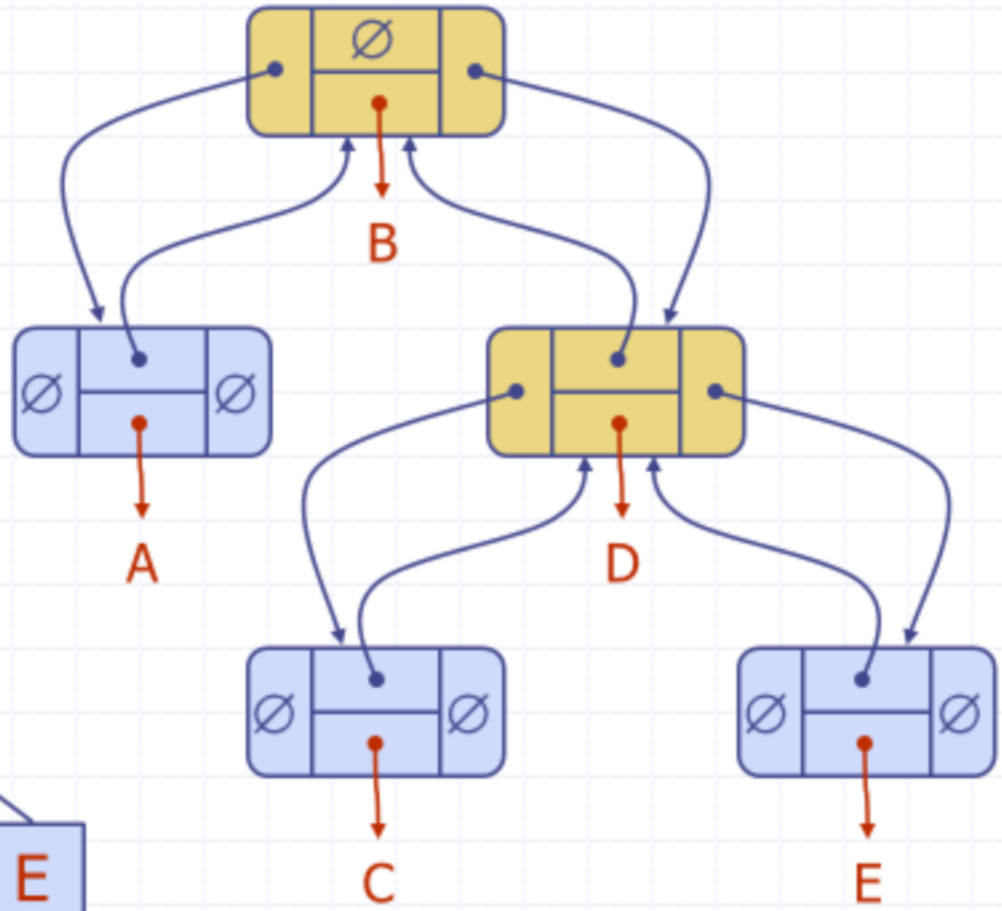
Note: Lot of recursive calls here. A stack is used to implement recursion.

H/w: Replace recursion with iterative procedure using stack. Will parent pointers help??

Linked Structure for Binary Trees

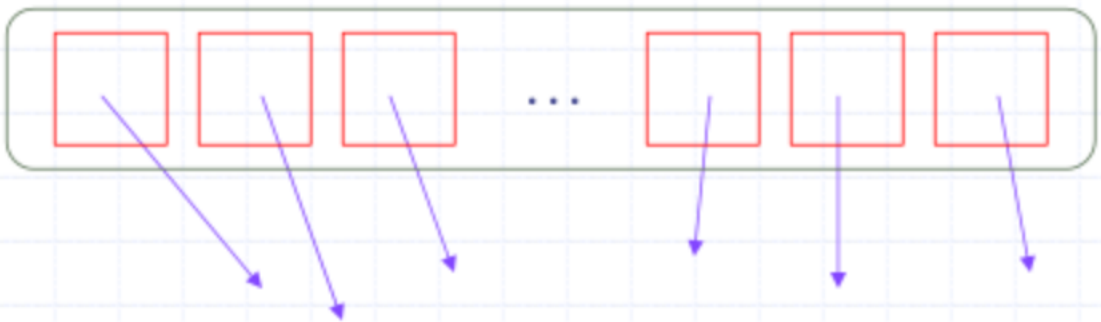
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT

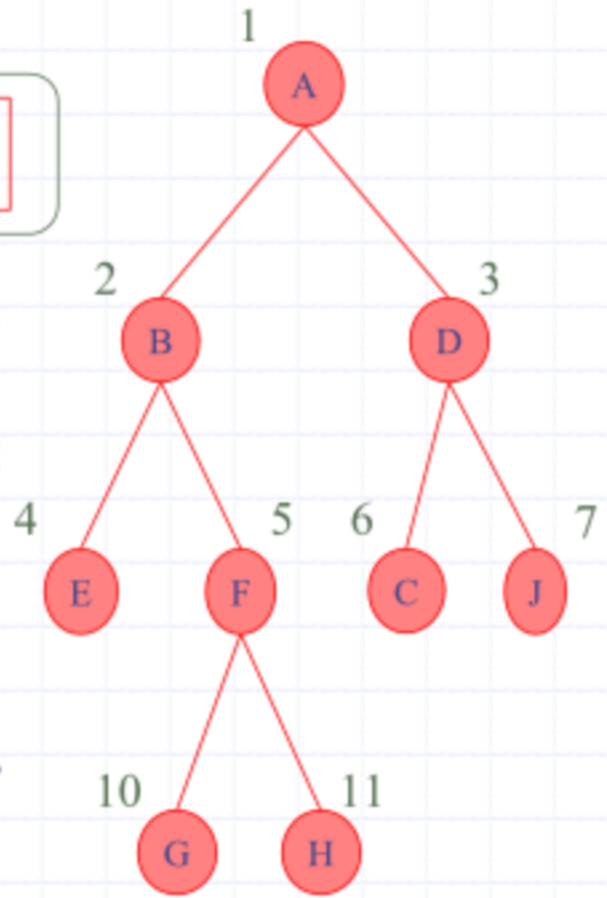




Array-Based Representation of Binary Trees

- nodes are stored in an array



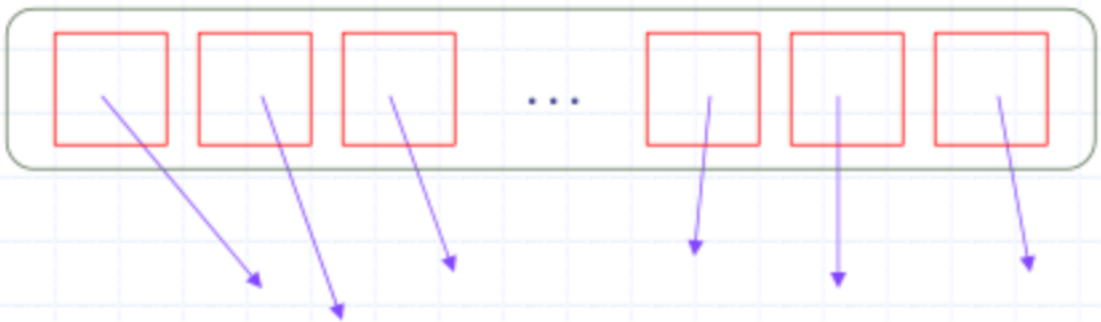


- let rank(node) be defined as follows:
  - rank(root) = 1
  - if node is the left child of parent(node), rank(node) = 2*rank(parent(node))
  - if node is the right child of parent(node), rank(node) = 2*rank(parent(node))+1

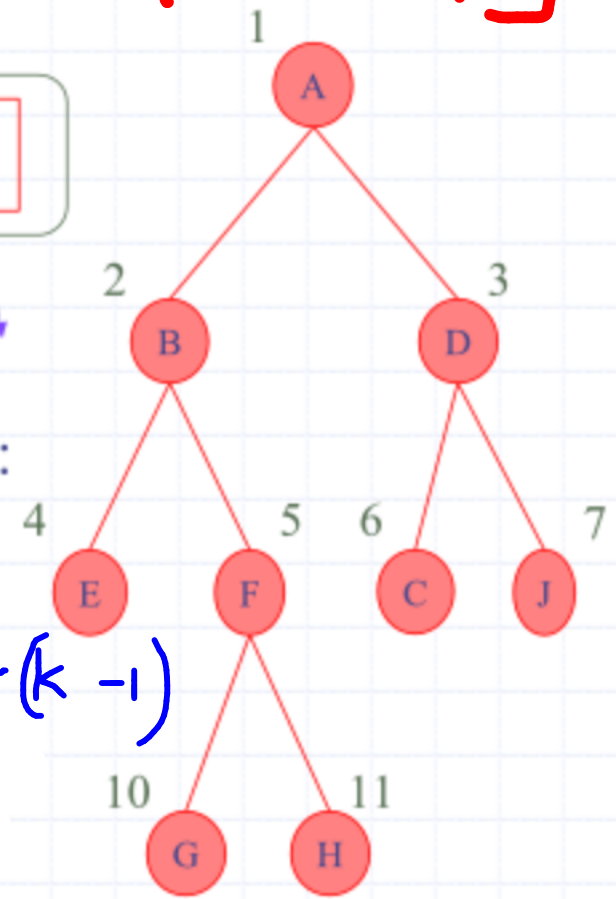
- For nodes that dont exist, you have NULL pointers.

Array-Based Representation of N-ary Trees [ie max # of children of a node = N]

- nodes are stored in an array



- let rank(node) be defined as follows:
  - rank(root) = 1
  - if node is the k^{th} child of parent(node),
    rank(node) = n * rank(parent(node)) + $(k-1)$



- For nodes that dont exist, you have NULL pointers.

For "sparse" n-ary trees:

Lots of missing nodes compared to balanced tree } Sparse cannot mean only 2 missing nodes!!!

Use same numbering scheme:
i^{th} child of k^{th} node: $nk+(i-1)$
& use use this as key into a HASHMAP

Priority Queues (PQ)

FIFO: Priority was "order" of insertion

PQ is generalisation of queue

Priority Queue ADT (§ 7.1.3)

- A priority queue stores a collection of entries
- Each **entry** is a pair (key, value)
- Main methods of the Priority Queue ADT
  - insert(k, x)
    inserts an entry with key k and value x
  - removeMin()
    removes and returns the entry with smallest key
- Additional methods
  - min()
    returns, but does not remove, an entry with smallest key
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

flight id

Eg: timestamp diff from current time

[Recall Extra Quiz problem 1]

Total Order Relations (§ 7.1.1)

- Keys in a priority queue can be arbitrary objects on which an order is defined
- Two distinct entries in a priority queue can have the same key

- Mathematical concept of total order relation $\leq$
  - Reflexive property: $x \leq x$
  - Antisymmetric property: $x \leq y \wedge y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y \wedge y \leq z \Rightarrow x \leq z$
- 2 elements must be comparable

Entry ADT (§ 7.1.2)

- An **entry** in a priority queue is simply a key-value pair
- Priority queues store entries to allow for efficient insertion and removal based on keys
- Methods:
  - key(): returns the key for this entry
  - value(): returns the value associated with this entry

- As a Java interface:

```
/**
 * Interface for a key-value
 * pair entry
**/
public interface Entry {
  public Object key();
  public Object value();
}
```

Comparator ADT (§ 7.1.2)

- A comparator encapsulates the action of comparing two objects according to a given total order relation
- A generic priority queue uses an auxiliary comparator
- The comparator is external to the keys being compared
- When the priority queue needs to compare two keys, it uses its comparator

- The primary method of the Comparator ADT:
  - compare(x, y): Returns an integer i such that $i < 0$ if $a < b$, $i = 0$ if $a = b$, and $i > 0$ if $a > b$; an error occurs if a and b cannot be compared.

Example Comparator

- Lexicographic comparison of 2-D points:

```
/** Comparator for 2D points under the
    standard lexicographic order. */
public class Lexicographic implements
    Comparator {
  int xa, ya, xb, yb;
  public int compare(Object a, Object b)
   throws ClassCastException {
    xa = ((Point2D) a).getX();
    ya = ((Point2D) a).getY();
    xb = ((Point2D) b).getX();
    yb = ((Point2D) b).getY();
    if (xa != xb)
        return (xb - xa);
    else
        return (yb - ya);
  }
}
```

- Point objects:

```
/** Class representing a point in the
    plane with integer coordinates */
public class Point2D        {
  protected int xc, yc; // coordinates
  public Point2D(int x, int y) {
    xc = x;
    yc = y;
  }
  public int getX() {
        return xc;
  }
  public int getY() {
        return yc;
  }
}
```