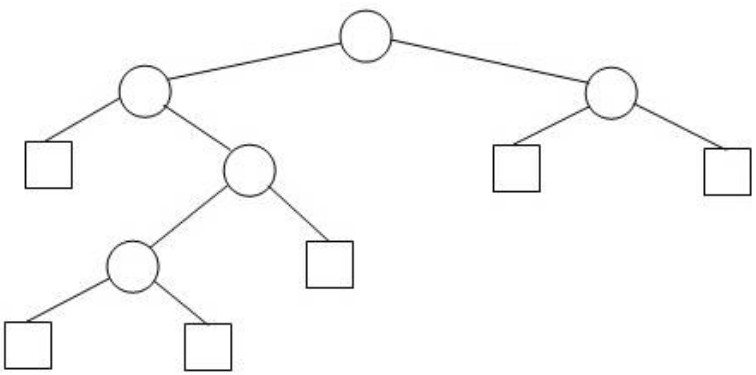
# CSSE 230



## Extended Binary Trees
## Recurrence relations

After today, you should be able to…
…explain what an extended binary tree is
…solve simple recurrences using patterns

# Reminders/Announcements

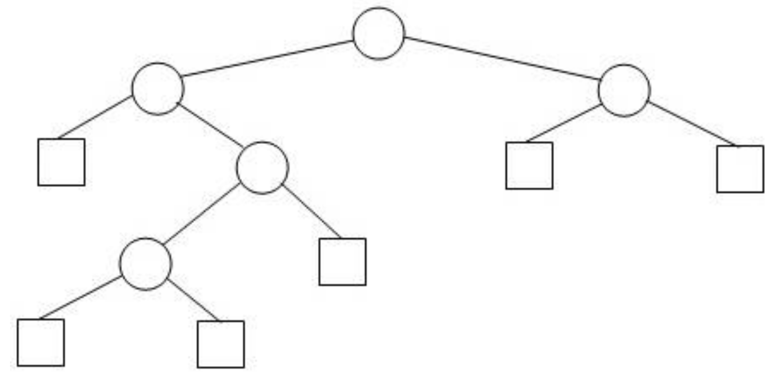
- Today:
  - Extended Binary Trees (on HW9)
  - Recurrence relations, part 1
- GraphSurfing Milestone 2
  - Two additional methods: shortestPath(T start, T end) and stronglyConnectedComponent(T key)
  - Tests on Living People subgraph of Wikipedia hyperlinks graph
  - Bonus problem: find a “challenge pair”
    - Pair with as-long-as-possible shortest path

# Extended Binary Trees (EBTs)

Bringing new life to Null nodes!

# An Extended Binary Tree (EBT) just has *null* external nodes as leaves

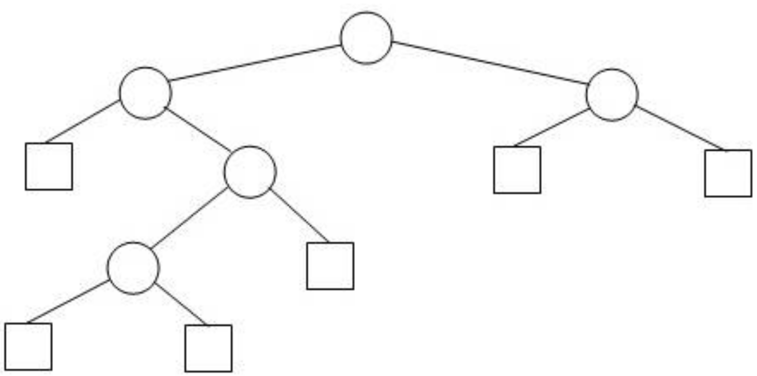




- Not a single NULL_NODE, but many NULL_NODEs
- An Extended Binary tree is either
  - an ***external (null) node***, or
  - an (**internal**) root node and two EBTs $T_L$ and $T_R$.
- We draw internal nodes as circles and external nodes as squares.
  - Generic picture and detailed picture.
- This is simply an alternative way of viewing binary trees, in which we view the external nodes as “places” where a search can end or an element can be inserted.

# A property of EBTs

- **Property** P(N): For any N>=0, any EBT with N internal nodes has _______ external nodes.
- **Prove by strong induction**, based on the recursive definition.
  - A notation for this problem: IN(T), EN(T)



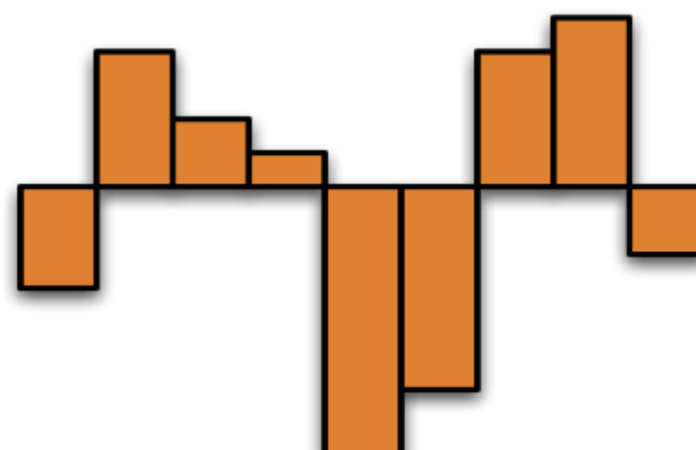
Hint (reminder): Find a way to relate the properties for larger trees to the property for smaller trees.

# Introduction to Recurrence Relations

A technique for analyzing recursive algorithms

# Recap: Maximum Contiguous Subsequence Sum problem

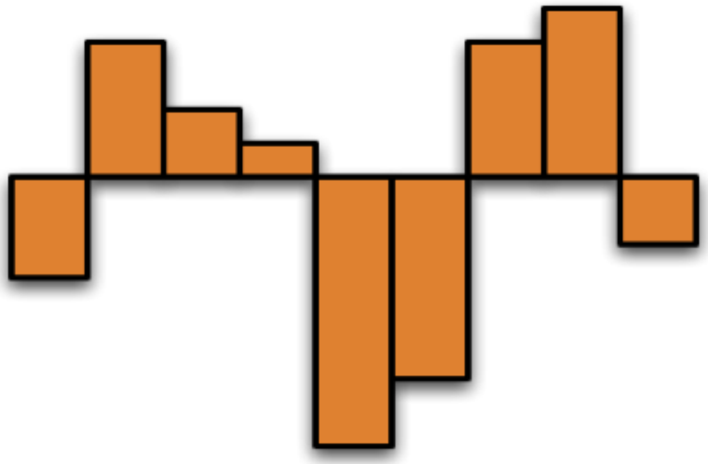
*Problem definition*: Given a non-empty sequence of $n$ (possibly negative) integers $A_1, A_2, \ldots, A_n$, find the maximum consecutive subsequence $S_{i,j} = \sum_{k=i}^{j} A_k$, and the corresponding values of $i$ and $j$.



# Divide and Conquer Approach

- Split the sequence in half
- Where can the maximum subsequence appear?

- Three possibilities :
  - entirely in the first half,
  - entirely in the second half, or
  - **begins** in the first half and **ends** in the second half



# This leads to a recursive algorithm

1. Using recursion, find the maximum sum of **first** half of sequence
2. Using recursion, find the maximum sum of **second** half of sequence
3. Compute the max of all sums that begin in the first half and end in the second half
    - (Use a couple of loops for this)
4. Choose the largest of these three numbers

```java
private static int maxSumRec( int [ ] a, int left, int right )
{
    int maxLeftBorderSum = 0, maxRightBorderSum = 0;
    int leftBorderSum = 0, rightBorderSum = 0;
    int center = ( left + right ) / 2;

    if( left == right )  // Base case
        return a[ left ] > 0 ? a[ left ] : 0;

    int maxLeftSum  = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );

    for( int i = center; i >= left; i-- )
    {
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }

    for( int i = center + 1; i <= right; i++ )
    {
        rightBorderSum += a[ i ];
        if( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }

    return max3( maxLeftSum, maxRightSum,
                 maxLeftBorderSum + maxRightBorderSum );
```

N = array size

What's the run-time?

```java
private static int maxSumRec( int [ ] a, int left, int right )
{
    int maxLeftBorderSum = 0, maxRightBorderSum = 0;
    int leftBorderSum = 0, rightBorderSum = 0;
    int center = ( left + right ) / 2;

    if( left == right )  // Base case
        return a[ left ] > 0 ? a[ left ] : 0;

    int maxLeftSum  = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );

    for( int i = center; i >= left; i-- )
    {
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }

    for( int i = center + 1; i <= right; i++ )
    {
        rightBorderSum += a[ i ];
        if( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }

    return max3( maxLeftSum, maxRightSum,
                 maxLeftBorderSum + maxRightBorderSum );
```

Runtime =
Recursive part +
non-recursive part

# Analysis?

- Write a **Recurrence Relation**
  - T(N) gives the run-time as a function of N
  - Two (or more) part definition:
    - Base case, like $T(1) = c$
    - Recursive case, like $T(N) = T(N/2) + 1$

So, what's the recurrence relation for the recursive MCSS algorithm?

```
private static int maxSumRec( int [ ] a, int left, int right )
{
    int maxLeftBorderSum = 0, maxRightBorderSum = 0;
    int leftBorderSum = 0, rightBorderSum = 0;
    int center = ( left + right ) / 2;

    if( left == right )  // Base case
        return a[ left ] > 0 ? a[ left ] : 0;

    int maxLeftSum  = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );

    for( int i = center; i >= left; i-- )
    {
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }

    for( int i = center + 1; i <= right; i++ )
    {
        rightBorderSum += a[ i ];
        if( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }

    return max3( maxLeftSum, maxRightSum,
                 maxLeftBorderSum + maxRightBorderSum );
```

Runtime = Recursive part + non-recursive part

```
private static int maxSumRec( int [ ] a, int left, int right )
{
    int maxLeftBorderSum = 0, maxRightBorderSum = 0;
    int leftBorderSum = 0, rightBorderSum = 0;
    int center = ( left + right ) / 2;

    if( left == right )  // Base case
        return a[ left ] > 0 ? a[ left ] : 0;

    int maxLeftSum  = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );

    for( int i = center; i >= left; i-- )
    {
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }

    for( int i = center + 1; i <= right; i++ )
    {
        rightBorderSum += a[ i ];
        if( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }

    return max3( maxLeftSum, maxRightSum,
                 maxLeftBorderSum + maxRightBorderSum );
```

Runtime = Recursive part + non-recursive part

$T(N) = 2T(N/2) + \theta(N)$

$T(1) = 1$

# Recurrence Relation, Formally

- An equation (or inequality) that relates the $n^{th}$ element of a sequence to certain of its predecessors (recursive case)
- Includes an initial condition (base case)
- **Solution**: A function of n.

- Similar to differential equation, but discrete instead of continuous
- Some solution techniques are similar to diff. eq. solution techniques

# Solve Simple Recurrence Relations

- One strategy: **look for patterns**

- Examples:
  **As class**:
  - $T(0) = 0$, $T(N) = 2 + T(N-1)$
  - $T(0) = 1$, $T(N) = 2\ T(N-1)$
  - $T(0) = T(1) = 1$, $T(N) = T(N-2) + T(N-1)$

  **On quiz:**
  - $T(0) = 1$, $T(N) = N\ T(N-1)$
  - $T(0) = 0$, $T(N) = T(N-1) + N$
  - $T(1) = 1$, $T(N) = 2\ T(N/2) + N$
    (just consider the cases where $N=2^k$)

# Next time: More solution strategies for recurrence relations

- Find patterns
- Telescoping
- Recurrence trees
- The master theorem

# GraphSurfing Work Time