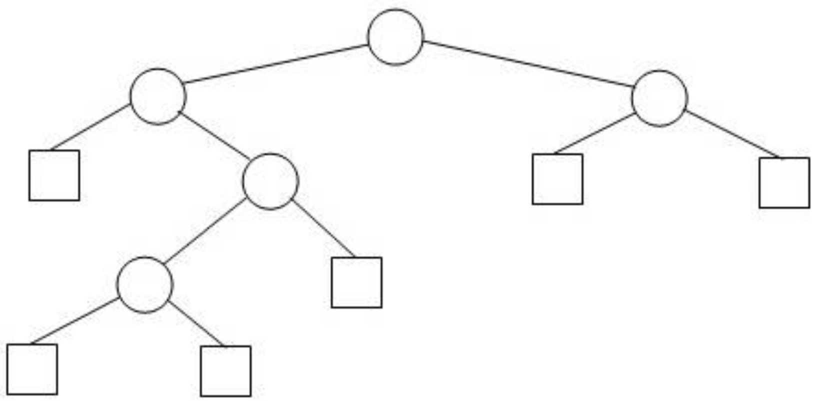


CSSE 230

How do EBTs help use analyze BST search?
What is a recurrence relation?

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

Reminders/Announcements

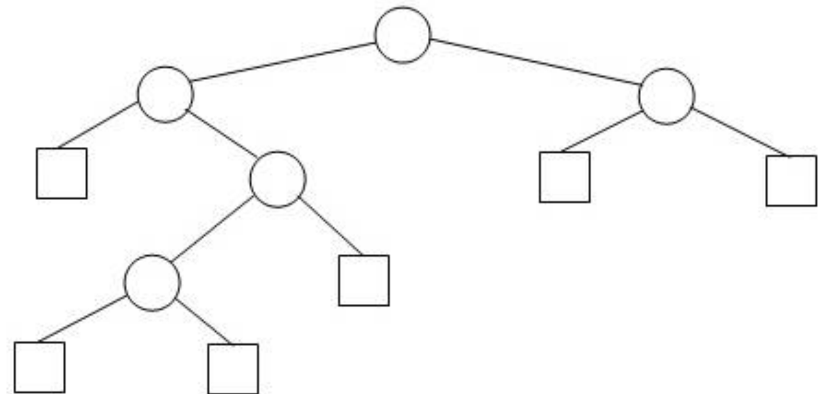
- Today:
  - Extended Binary Trees (on HW9)
  - *Average-case* analysis of successful search on a (naïve) BST
  - Recurrence relations, part 1
- GraphSurfing Milestone 2
  - Two additional methods: shortestPath(T start, T end) and stronglyConnectedComponent(T key)
  - Tests on Living People subgraph 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

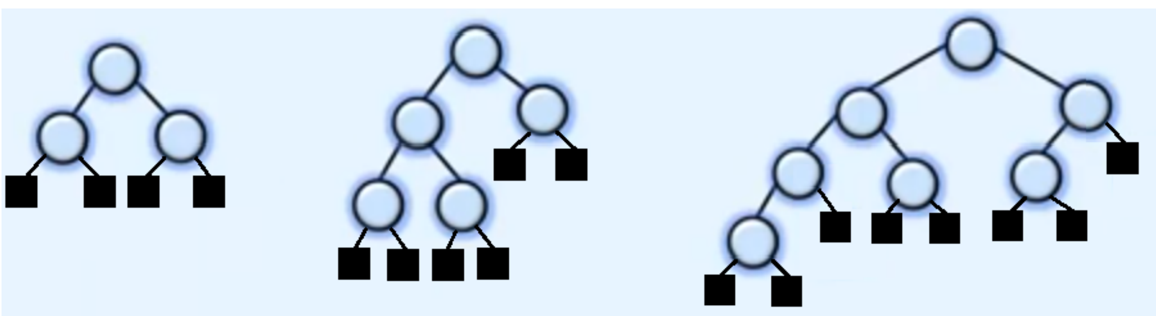
- Not a single NULL_NODE, but many NULL_NODEs

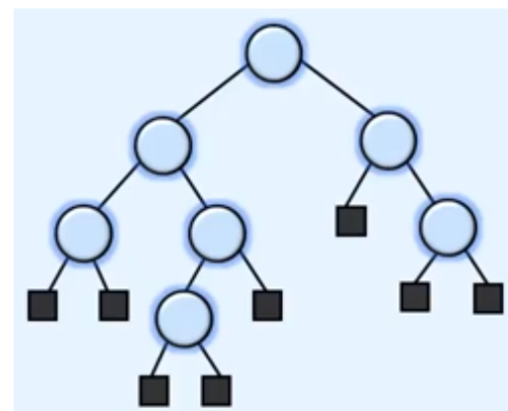


- An Extended Binary tree is either
  - an ***external (null) node***, or
  - an (**internal**) root node and two EBTs T_L and T_R, that is, all nodes have 2 children
- Convention.
  - Internal nodes are circles
  - External nodes are squares
- This is simply an alternative way of viewing binary trees: external nodes are "places" where a search can end or an element can be inserted – for a BST, what legal values could eventually be inserted at an external node?

A property of EBTs

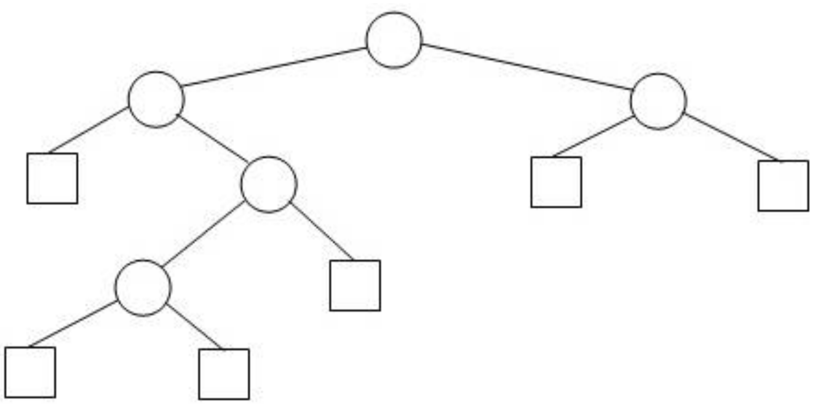
- **Property** P(N): For any N $\geq$ 0, any EBT with N internal nodes has ________ external nodes.
- Use example trees below to come up with a formula, let:
  - EN(T) = external nodes
  - IN(T) = internal nodes





A property of EBTs

- **Property** P(N): For any $N \geq 0$, any EBT with N internal nodes has N+1 external nodes.
- **Prove by strong induction**, based on the recursive definition.
  - A notation for this problem: IN(T), EN(T)

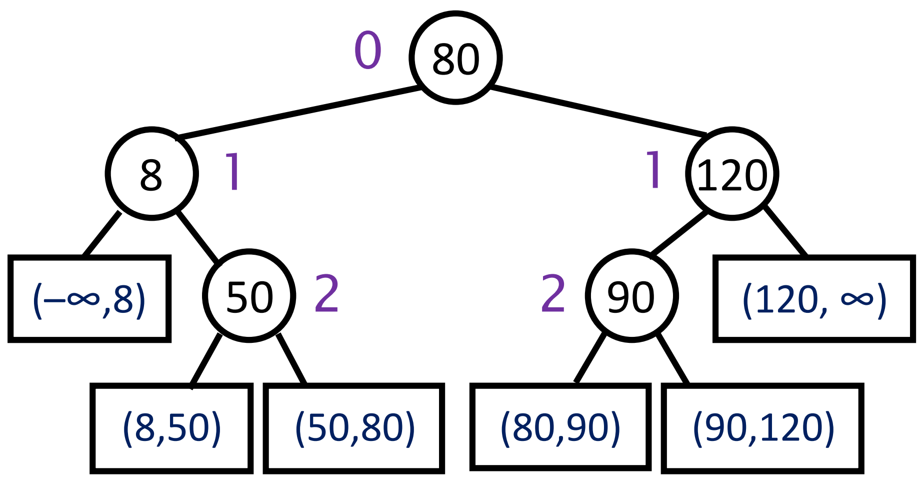


Hint (reminder): Find a way to relate the properties for larger trees to the property for smaller trees.

Analyzing BST Successful Search

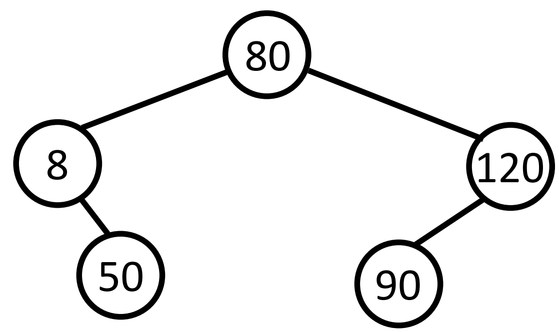
- Define *internal path length*: the sum of depths of internal nodes in an EBT.
- How does it relate to the average-case running time of successful search on a BST?

EBT:



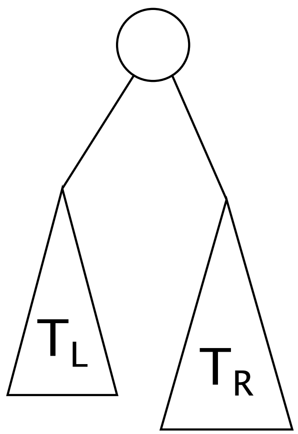
IPL = 0 + 1 + 1 + 2 + 2 = 6

BST:



Average-Case Analysis of Successful Search on a BST

- Idea: find expected IPL(T), for a tree T of size N.
  - Key point: How does IPL(T) relate to IPL(T_L) and IPL(T_R)?



$$IPL(T) = N - 1 + IPL(T_L) + IPL(T_R)$$

  - Take expected value of both sides

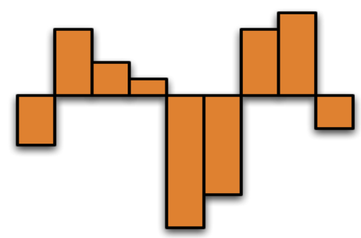
- Then, expected runtime of successful search is [expected IPL] / N.

Introduction to Recurrence Relations

A technique for analyzing recursive algorithms

Recall: 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.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

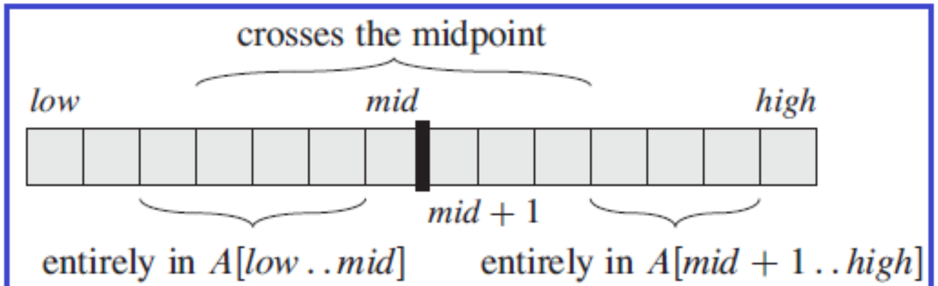
maximum subarray

Chapter 4 - Cormen 3e

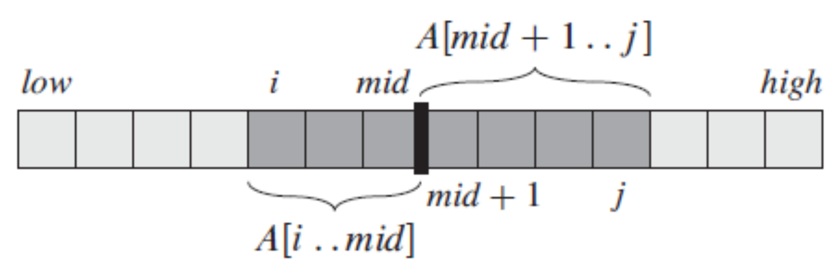
New MCSS Algorithm: Using Divide & Conquer

- 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



Chapter 4 - Cormen 3e



Chapter 4 - Cormen 3e

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

```
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?

General Form – Recurrence

$T(n) = aT(n/b) + f(n)$

- a = # of subproblems
- n/b = size of subproblem
- $f(n) = D(n) + C(n)$
- $D(n)$ = time to divide problem before recursion
- $C(n)$ = time to combine after recursion

```
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

```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

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

$T(1) = 1$

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**
  - Forward substitution
  - Backward substitution

- Examples:

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

On quiz:
1. $T(0) = 1, T(N) = N\ T(N-1)$
2. $T(0) = 0, T(N) = T(N-1) + N$
3. $T(1) = 1, T(N) = 2\ T(N/2) + N$
   (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