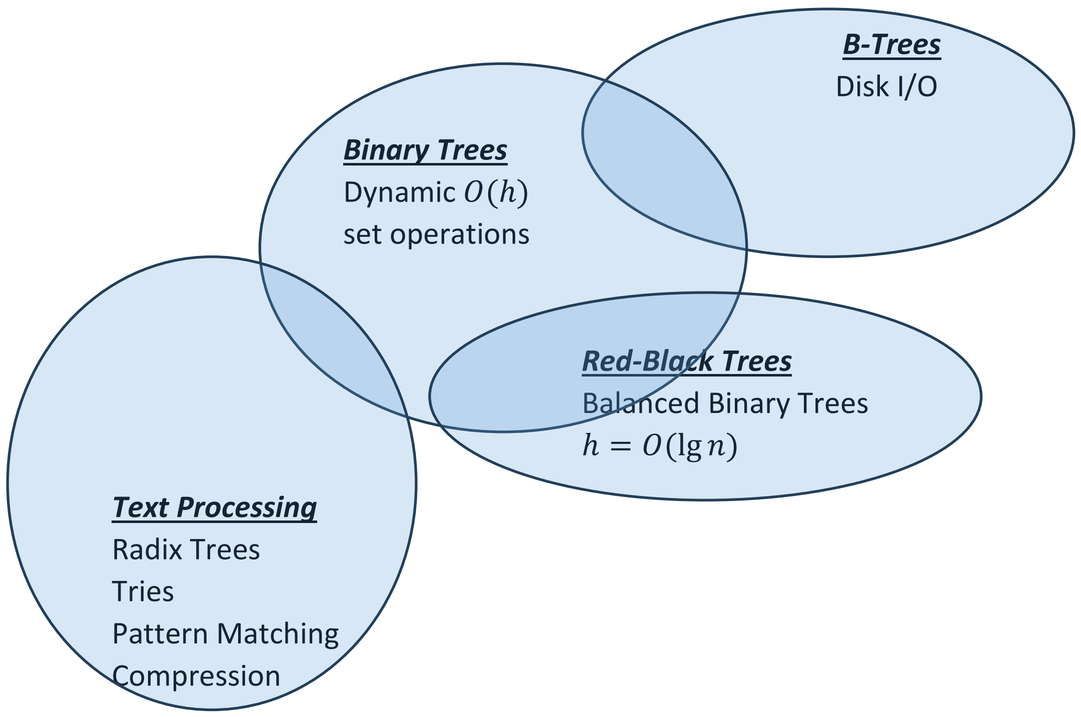
CS 315 - Day 23, Trees

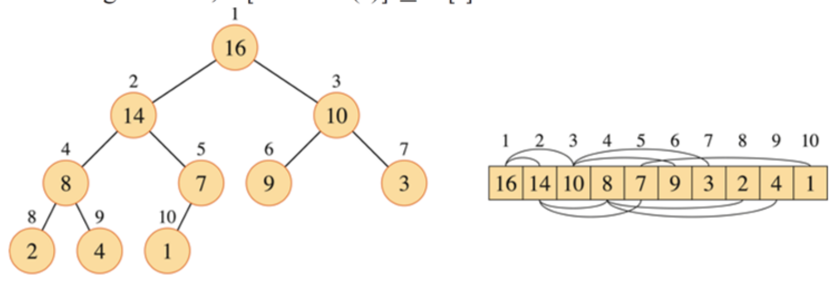


These are some of the next topics we will be covering, and how they overlap.

First, a brief review, what is the Heap Property?

Of a max-heap in array with *heap-size* $= 10$.

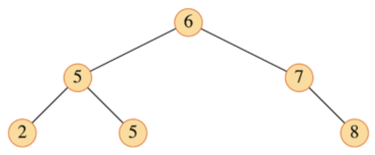
- For max-heaps (largest element at root), ***max-heap property:*** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \geq A[i]$.
- For min-heaps (smallest element at root), ***min-heap property:*** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \leq A[i]$.



Next, we reacall the Binary-Search Tree (BST) Property:

Stored keys must satisfy the ***binary-search-tree property***.

- If y is in left subtree of x, then $y.key \leq x.key$.
- If y is in right subtree of x, then $y.key \geq x.key$.



The binary-search-tree property allows us to print keys in a binary search tree in order, recursively, using an algorithm called an ***inorder tree walk***. Elements are printed in monotonically increasing order.

Search trees

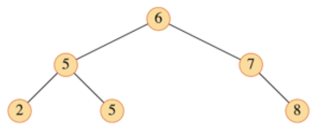
- Data structures that support many dynamic-set operations.
- Can be used as both a dictionary and as a priority queue.
- Basic operations take time proportional to the height of the tree.
  - For complete binary tree with n nodes: worst case $\Theta(\lg n)$.
  - For linear chain of n nodes: worst case $\Theta(n)$.

Binary search trees are an important data structure for dynamic sets.

- Accomplish many dynamic-set operations in $O(h)$ time, where h = height of tree.
- As in Section 10.3, represent a binary tree by a linked data structure in which each node is an object.
- $T.root$ points to the root of tree T.
- Each node contains the attributes
  - *key* (and possibly other satellite data).
  - *left*: points to left child.
  - *right*: points to right child.
  - p: points to parent. $T.root.p$ = NIL.

How INORDER-TREE-WALK works:

- Check to make sure that x is not NIL.
- Recursively print the keys of the nodes in x's left subtree.
- Print x's key.
- Recursively print the keys of the nodes in x's right subtree.



```
INORDER-TREE-WALK(x)
1  if $x \neq$ NIL
2      INORDER-TREE-WALK($x.left$)
3      print $x.key$
4      INORDER-TREE-WALK($x.right$)
```

Time

Intuitively, the walk takes $\Theta(n)$ time.

An Inorder traversal of a BST is always in sorted order.

Show the Inorder, Preorder (6, 5, 2, 5, 7, 8), and Postorder (2, 5, 5, 8, 7, 6) traversals for the above tree (***"trick" w/outline***), then give a different tree, satisfying the BST, with the same nodes as above.

Searching a BST

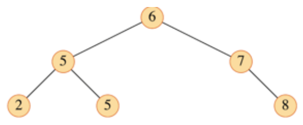
Recursive Search

```
TREE-SEARCH(x, k)
1  if $x$ == NIL or $k$ == $x.key$
2      return $x$
3  if $k < x.key$
4      return TREE-SEARCH($x.left$, $k$)
5  else return TREE-SEARCH($x.right$, $k$)
```

Initial call is TREE-SEARCH($T.root$, k).

Iterative version

The iterative version unrolls the recursion into a **while** loop. It's usually more efficient than the recursive version, since it avoids the overhead of recursive calls.



```
ITERATIVE-TREE-SEARCH(x, k)
1  while $x \neq$ NIL and $k \neq x.key$
2      if $k < x.key$
3          $x = x.left$
4      else $x = x.right$
5  return $x$
```

The Recursive algorithm visits nodes on a downward path from the root, so the running time is on the order of the height of the BST, or $O(h)$.

<u>Minimum and Maximum of a BST</u> (aka, “always go left,” or “always go right”)

The binary-search-tree property guarantees that

- the minimum key of a binary search tree is located at the leftmost node, and
- the maximum key of a binary search tree is located at the rightmost node.

Traverse the appropriate pointers (*left* or *right*) until NIL is reached. In the following procedures, the parameter x is the root of a subtree. The first call has $x = T.root$.

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

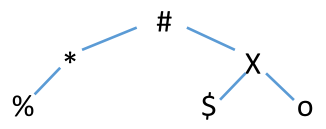
Time: Both procedures visit nodes that form a downward path from the root to a leaf. Both procedures run in $O(h)$ time, where h is the height of the tree.

<u>In-class w/***assigned groups of four (4)***, write 0. and 1. on board as ground rule(s):</u>

0. No devices.
1. Intro: Name, POE, Icebreaker.
2. Pick someone on the group who will report out.
3. Given the following orderings, construct the tree:

    Inorder: `%, *, #, $, X, o`

    Preorder: `#, *, %, X, $, o`



4. In your group, create a tree, and then choose someone to write its (1) Inorder and (2) one of its Preorder or Postorder traversals on the board.
5. As groups, construct the other trees given.

In doing operations on BSTs, we introduce the idea of ***Successor*** and ***Predecessor*** (symmetric to Successor):

Assuming that all keys are distinct, the successor of a node x is the node y such that $y.key$ is the smallest key $> x.key$. We can find x's successor based entirely on the tree structure. No key comparisons are necessary. If x has the largest key in the binary search tree, then x's successor is NIL.

There are two cases:

1. If node x has a non-empty right subtree, then x's successor is the minimum in x's right subtree.
2. If node x has an empty right subtree, notice that:
    - As long as we move to the left up the tree (move up through right children), we're visiting smaller keys.
    - x's successor y is the node that x is the predecessor of (x is the maximum in y's left subtree).

Give the Successor for each node in the above tree:

Node	Successor
%	*
*	#
#	$
X	o
$	X
o	NIL

Easily, the Successor of a node in a BST is the node directly after it in an Inorder Tree Traversal.

Algorithmically, the structure of a BST allows you to determine the successor of a node without comparing key values of any nodes.

If the node ***<u>has</u>*** a right subtree, its successor is the leftmost node in that right subtree.

If it doesn't have a right subtree, its successor is its lowest ancestor whose left child is also an ancestor.

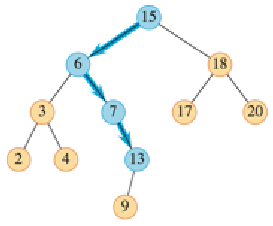
```
TREE-SUCCESSOR($x$)
if $x.right \neq$ NIL
    return TREE-MINIMUM($x.right$)    // leftmost node in right subtree
else // find the lowest ancestor of $x$ whose left child is an ancestor of $x$
    $y = x.p$
    while $y \neq$ NIL and $x == y.right$
        $x = y$
        $y = y.p$
    return $y$
```

TREE-PREDECESSOR is symmetric to TREE-SUCCESSOR.

In your groups, <u>apply the algorithm</u> to the following:

Example

Find successor of 15, 6, and 4.



Insertion is fairly straightforward, and its runtime depends on the height of the tree:

```
TREE-INSERT($T, z$)
$x = T.root$              // node being compared with $z$
$y =$ NIL                 // $y$ will be parent of $z$
while $x \neq$ NIL        // descend until reaching a leaf
    $y = x$
    if $z.key < x.key$
        $x = x.left$
    else $x = x.right$
$z.p = y$                 // found the location—insert $z$ with parent $y$
if $y ==$ NIL
    $T.root = z$          // tree $T$ was empty
elseif $z.key < y.key$
    $y.left = z$
else $y.right = z$
```

- To insert value v into the binary search tree, the procedure is given node z, with $z.key$ already filled in, $z.left =$ NIL, and $z.right =$ NIL.
- Beginning at root of the tree, trace a downward path, maintaining two pointers.
  - Pointer x: traces the downward path.
  - Pointer y: "trailing pointer" to keep track of parent of x.
- Traverse the tree downward by comparing $x.key$ with $z.key$, and move to the left or right child accordingly.
- When x is NIL, it is at the correct position for node z.
- Compare $z.key$ with $y.key$, and insert z at either y's *left* or *right*, appropriately.