CECS 274: Data Structures
Binary Trees

Oscar Morales Ponce

California State University Long Beach

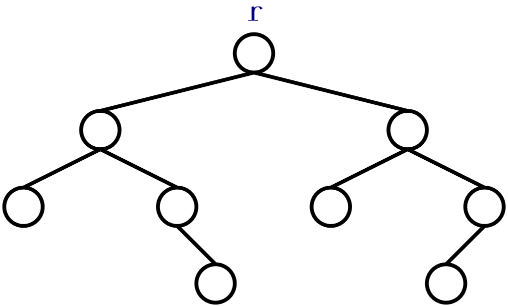
Example

- Mathematical Expressions
- We can check if the following expression is valid using a Stack (Array-based, Linked-List based)

$$(a+b)*(c*d)+(a+(b*(c/d))$$

- We can replace the terms with the actual values efficiently using a USet (ChainedHashTable)
  $a = 5413.13, b = 243.12, c = 4212.12, d = 421.3$
- The next natural question is how it can be evaluated.

Binary Tree



Binary Trees

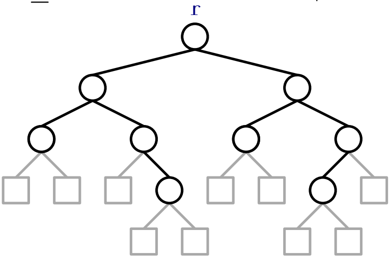
- A *binary tree* is a connected, undirected, finite graph with no cycles, and no vertex of degree greater than three.
- For most computer science applications, binary trees are *rooted:*
  - A special node, r, of degree at most two is called the *root* of the tree.
  - For every node, $u \neq r$, the second node on the path from u to r is called the *parent* of u.
  - Each of the other nodes adjacent to u is called a *child* of u.
  - Most of the binary trees we are interested in are *ordered*, so we distinguish between the *left child* and *right child* of u.

Terminology

- The depth of a node, u, is the length of the path from u to the root of the tree.
- If a node, w, is on the path from u to r, then w is called an *ancestor* of u and u a *descendant* of w.
- The *subtree* of a node, u, is the binary tree that is rooted at u and contains all of u's descendants.
- The *height* of a node, u, is the length of the longest path from u to one of its descendants.
- The *height* of a tree is the height of its root.
- A node, u, is a *leaf* if it has no children.

External nodes

- We sometimes think of the tree as being augmented with *external nodes*.
- Any node that does not have a left child has an external node as its left child, and, correspondingly, any node that does not have a right child has an external node as its right child
- It is easy to verify, by induction, that a binary tree with $n \geq 1$ real nodes has $n + 1$ external nodes.



Auxiliary Structure Node

- Node
  - State variables:
    - $left$: Points to the left node
    - $right$: Points to the right node
    - $parent$: Points to the parent node
    - x: The data to store
    - v: The value to store. It is used in the BinarySearchTree
  - Invariant: If $left$ is not present, $left = nil$
    If $right$ is not present, $right = nil$
    If $parent$ is not present, $parent = nil$

Auxiliary Structure Node (auxiliar methods)

$set\_val(key, value)$

$x = key$

$v = value$

$insert\_left()$

$left = BinaryTree.Node('')$

$left.parent = this$

return $left$

$insert\_right()$

$right = BinaryTree.Node('')$

$right.parent = this$

return $right$

BinaryTree: The Basics

- BinaryTree
  - State variables:
    - r: the root of the binary tree
  - Invariant: $r = nil$ if the BinaryTree is empty

initialize()
  $r \leftarrow nil$

BinaryTree: $depth(u)$

- $depth(u)$: Returns the depth of the node u in the BinaryTree

```
depth(u)
    Check preconditions
    Count the steps on the path from u to the root
```

BinaryTree: $depth(u)$

```
depth(u)
    if u = nil return −1
    d ← 0
    while (u ≠ r) do
        u ← u.parent
        d ← d + 1
    return d
```

- How many operations?

BinaryTree: $size(u)$ (Recursive)

- $size()$: Return the number of elements in the BinaryTree

size(u)
  If u is nil, then return 0
  Else return 1 plus the size of left and right subtree

- Recursive methods solve a problem where the solution depends on solutions of smaller instances of the same problem

BinaryTree: $size(u)$ (Recursive)

size(u)
    if $u = nil$ **then return** 0
    return $1 + \text{size}(u.left) + \text{size}(u.right)$

- How many operations?

BinaryTree: $in\_order(u)$

- To visit the binary search in order, first we visit $left$, $root$ and then $right$
- $in\_order(u)$: $left, root, right$

$in\_order(u)$
    if $u.left = nil$ **then** $in\_order(u.left)$
    # Visit u
    if $u.right = nil$ **then** $in\_order(u.right)$

BinaryTree: $pre\_order(u)$

- How should we traverse a binary tree to create a copy?
- Copy first $root$, then $left$ and then $right$
- $pre\_order(u)$: $root, left, right$

$pre\_order(u)$

  # Visit u

  if $u.left = nil$ **then** $pre\_order(u.left)$

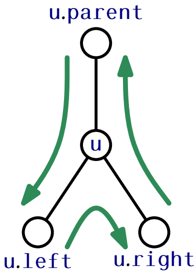
  if $u.right = nil$ **then** $pre\_order(u.right)$

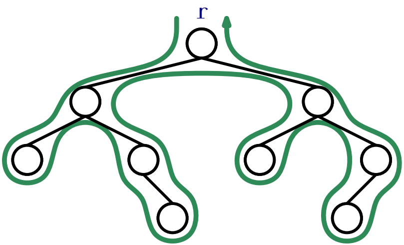
BinaryTree: $post\_order(u)$

- How should we traverse a binary tree to delete a tree?
- delete first $left$, then $right$ and then $root$ nodes
- $post\_order(u)$: $left, right, root$

$post\_order(u)$
  if $u.left = nil$ **then** $post\_order(u.left)$
  if $u.right = nil$ **then** $post\_order(u.right)$
  # Visit u

Traversing a tree in order without recursion





Traversing a tree

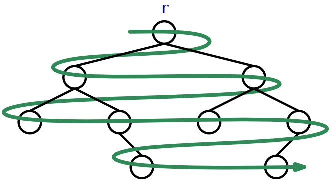
traverse2()

Let u be the node being visited. Initially, the root
Let $prev$ be the previous node being visited. Initially, nil
While there are no nodes without being visited
If we arrive at a node u from $u.parent$
Let nxt be the next node to visit: $left$ if exists
otherwise $right$ otherwise $parent$
Else if we arrive at a node u from $u.left$
Let nxt be the next node to visit: $right$ if exists
otherwise $parent$
Else if we arrive at a node u from $u.right$
Let nxt be the next the $parent$
Update $prev$ with u
Update u with $next$

Traversing a tree

```
traverse2()
    u ← r
    prv ← nil
    while u ≠ nil do
        if prv = u.parent then
            # visit u for first time
            if u.left ≠ nil then nxt ← u.left
            else if u.right ≠ nil nxt ← u.right
            else nxt ← u.parent
        else if prv = u.left
            if u.right ≠ nil then nxt ← u.right
            else nxt ← u.parent
        else
            nxt ← u.parent
        prv ← u
        u ← nxt
```

Breadth-first traversal



- The nodes are visited level-by-level starting at the root and moving down, visiting the nodes at each level from left to right

Breadth-first traversal

```
bf_traverse()
    Insert the root in a queue
    While the queue is not empty
        Remove the next node in the queue
        Insert the left child if it exits
        Insert the right child if it exits
```

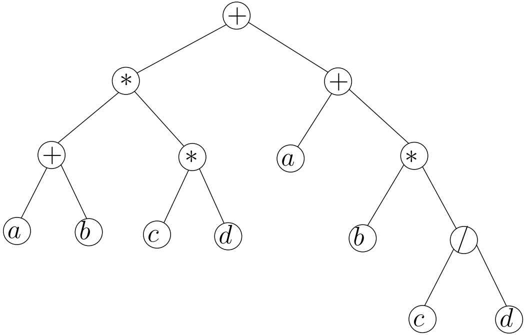
Breadth-first traversal

```
bf_traverse()
    q ← ArrayQueue()
    if r ≠ nil then q.add(r)
    while q.size() > 0 do
        u ← q.remove()
        if u.left ≠ nil then q.add(u.left)
        if u.right ≠ nil then q.add(u.right)
```

- How many operations?

- Parse trees can be used to represent real-world constructions like sentences or mathematical expressions.

$$(((a + b) * (c * d)) + (a + (b * (c/d))))$$



Building a Parse Tree from the Fully Parenthesized Mathematical Expression

- Breakup the expression in tokens: left parentheses, right parentheses, operator and operands.

```
build_parse_tree(exp)
    Let t be a binary tree
    Let u be the root
    While there are not more tokens in exp
        If token is '(' then
            Add a left child to u and let u = u.left
        If token is either '+, -, /, *' then
            u.x = token.
            Add a right child to u and let u = u.right
        If token is an operand then
            u.x = token and let u = u.parent
        If token is ')' then
            Let u = u.parent
    return t
```

Evaluate from a Parse Tree

$\_evaluate(node)$
    If $node.left$ and $node.right$ are not nil
        let $op = node.x$
        return $evaluate(node.left)\ op\ evaluate(node.left)$
    else $evaluate(node.x)$

$evaluate(tree)$
    return $\_evaluate(tree.r)$

Evaluate from a Parse Tree

```
import operator as oper

def _evaluate(self, node):
  op = { '+':operadd, '-':oper.sub, '*':oper.mul, '/':oper.truediv}
  if node.left != None and node.right != None:
      fn = op[node.x]
      return fn(self._evaluate(node.left), self._evaluate(node.right))
   else:
       t = self.dict.find(node.x)
       if t != None: return t
       return node.x
```

Autocompletion Feature

- In many web services while writing, the system shows a suggestion to autocomplete
- Using lists takes linear time in the worst case to find the first item that matches the prefix.
- In unsorted dictionaries, we can search the complete word in expected constant time, but finding the prefix takes also linear time in the worst case
- Is there a more efficient data structure to find the prefix?
- Sorted Dictionaries are the solutions.

Lexicographical Order

- Let $\omega = \alpha\beta\gamma$ and $\omega' = \alpha\delta\epsilon$

$$\omega < \omega' \text{ if and only if } \beta < \delta$$

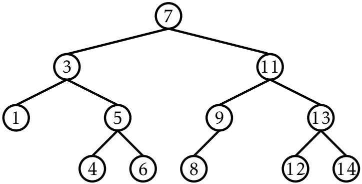
- For example $\omega_1 =$distraction and $\omega_2 =$distributed
  - $\omega_1 = \alpha\beta\gamma$ where $\alpha =$dist, $\beta =$a and $\gamma =$ctoin
  - $\omega_2 = \alpha\delta\epsilon$ where $\alpha =$dist, $\delta =$r and $\epsilon =$ibuted

$$\omega_1 < \omega_2$$

BinarySearchTree: An Unbalanced Binary Search Tree

- BinarySearchTree:
  - State variables:
    - n: Number of elements in the binary search tree.
    - r: The root of the tree
  - Invariant:
    - r is nil if $n = 0$
    - For every node u in the binary search tree

$$u.left.x < u.x < u.right.x$$



BinarySearchTree: $find\_eq(x)$

- $find\_eq(x)$: Return the node that contains the value x if it exits, or nil otherwise

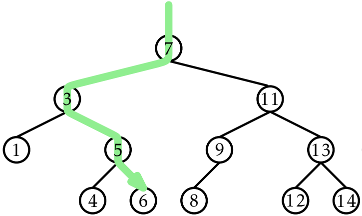
```
find_eq(x)
    Let w be the root
    While we don't find the value or w is nil
        If x is less than w.x, then visit the left child
        If x is greater than w.x, then visit the right child
        If x is equal w.x, then we found it!
    The value is not present
```

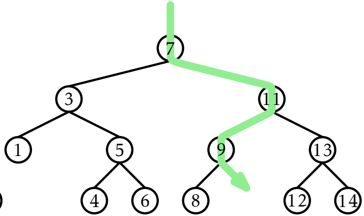
BinarySearchTree: $find\_eq(x)$

```
find_eq(x)
    w ← r
    while w ≠ nil do
        if x < w.x then
            w ← w.left
        else if x > w.x
            w ← w.right
        else
            return w.v
    return nil
```

- How many operations?

BinarySearchTree: Example $find\_eq(6)$, $find\_eq(10)$





BinarySearchTree: $find(x)$

- $find(x)$: Return the node that contains the value x if it exits. Otherwise it returns the node with the smallest value greater than x.
  - If we look at the last node, u, at which $x < u.x$, we see that $u.x$ is the smallest value in the tree greater than x.
  - Similarly, the last node at which $x > u.x$ contains the largest value less than x.
  - Therefore, we keep track of the last node, z, at which $x < u.x$

BinarySearchTree: $add\_child(p, u)$

- $add\_child(p, u)$: Add the node u as a child of p.

$\mathrm{add\_child}(p, u)$
   Check Preconditions
   Check Invariants
   Set p as the parent of u
   Increment n by one

BinarySearchTree: $add\_child(p, u)$

- $add\_child(p, u)$: Add the node u as a child of p.

add_child(p, u)
    If p is nil, then u becomes the root
    Otherwise if $u.x < p.x$, then u is the left child of p
    Otherwise if $u.x > p.x$, then u is the right child of p
    Otherwise if $u.x = p.x$, then nothing to do
    Set p as the parent of u
    Increment n by one

```
add_child(p, u)
    if p = nil then
        r ← u # inserting into empty tree
    else
        if u.x < p.x then
            p.left ← u
        else if u.x > p.x
            p.right ← u
        else
            return false # u.x is already in the tree
        u.parent ← p
    n ← n + 1
    return true
```

BinarySearchTree: $find\_last(x)$

- $find\_last(x)$: Return the node w such that $w.x = x$ if it exists, otherwise, return the previous node

```
find_last(x)
    w ← r
    prev ← nil
    while w ≠ nil do
        prev ← w
        if (x < w.x) then
            w ← w.left
        else if (x > w.x)
            w ← w.right
        else
            return w
    return prev
```

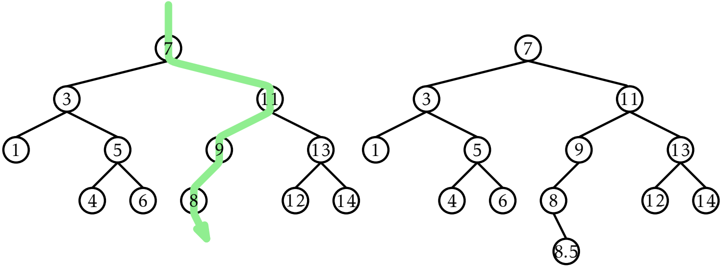
BinarySearchTree: $add(x, v)$

- $add(x, v)$: Add a node with x and value v in the binary search tree
  - Let p be the closest node to x
  - Insert a child to the binary search tree using $add\_child(p, new\_node(x, v))$

$\mathrm{add}(x)$
  $p \leftarrow \mathrm{find\_last}(x)$
  return $\mathrm{add\_child}(p, \mathrm{new\_node}(x, v))$

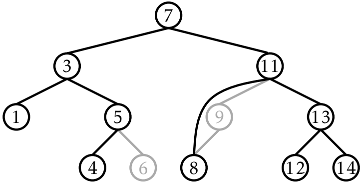
- How many operations?

BinarySearchTree: $add(8.5, Nill)$



BinarySearchTree: Node removal

- If u is a leaf, then we can just detach u from its parent.
- If u has only one child, then we can splice u from the tree by having $u.parent$ adopt u's child



BinarySearchTree: $splice(u)$

```
splice(u)
    if u.left ≠ nil then
        s ← u.left
    else
        s ← u.right
    if u = r then
        r ← s
        p ← nil
    else
        p ← u.parent
        if p.left = u then
            p.left ← s
        else
            p.right ← s
    if s ≠ nil then
        s.parent ← p
    n ← n − 1
```

BinarySearchTree: $remove\_node(u)$

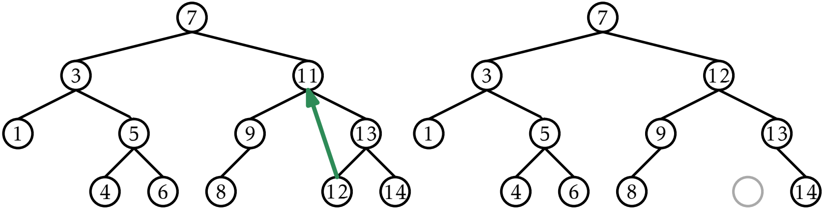
- $remove\_node(u)$: Remove the node u from the binary search tree

```
remove_node(u)
    If u has at most one child, then call splice(u)
    Otherwise find w that has at most one child such that
        w.x > x and w.x is the smallest
    Replace u.x with w.x
    Call splice(w)
```

BinarySearchTree: $remove\_node(u)$

```
remove_node(u)
    if u.left = nil or u.right = nil then
        splice(u)
    else
        w ← u.right
        while w.left ≠ nil do
            w ← w.left
        u.x ← w.x
        splice(w)
```

BinarySearchTree: $remove\_node(u)$



BinarySearchTree: $remove(x)$

- $remove(x)$: Remove the node with value x in the binary search tree
  - Let w be the node that $find\_eq(x)$ returns.
  - Call $remove\_node(w)$

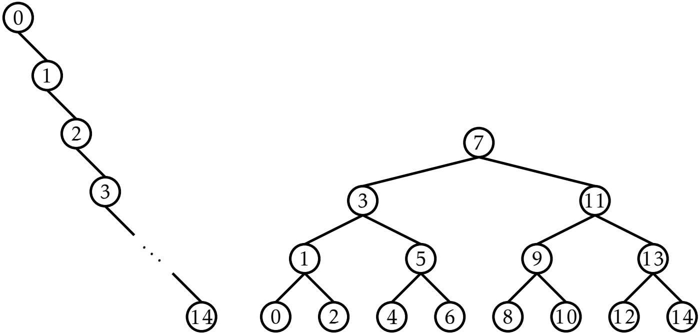
$$
\begin{array}{l}
\text{remove}(x) \\
\quad p \leftarrow \text{find\_eq}(x) \\
\quad remove\_node(w)
\end{array}
$$

Summary

- Theorem
  BinarySearchTree implements the SSet interface and supports the operations $\mathrm{add}(x)$, $\mathrm{remove}(x)$, *and* $\mathrm{find}(x)$ *in* $O(n)$ *time per operation.*
  - The problem with the BinarySearchTree structure is that it can become *unbalanced*.

Random Binary Trees

- Consider the binary trees obtained from inserting $\langle 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 \rangle$ in any random order



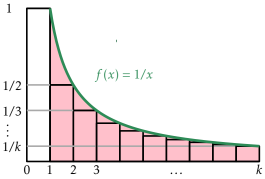
- Only one gives the left tree and $21,964,800$ that give the balanced tree

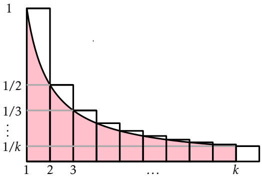
Harmonic Function

$$H_k = 1 + 1/2 + 1/3 + \cdots + 1/k \ .$$

The harmonic number H_k has no simple closed form, but it is very closely related to the natural logarithm of k. In particular,

$$\ln k < H_k \leq \ln k + 1 \ .$$





Random Binary Trees

Lemma

In a random binary search tree of size n, the following statements hold:

1. *For any $x \in \{0, \ldots, n-1\}$, the expected length of the search path for x is $H_{x+1} + H_{n-x} - O(1)$ (the number of elements in the tree less than or equal to x and the number of elements in the tree greater than or equal to x.)*
2. *For any $x \in (-1, n) \setminus \{0, \ldots, n-1\}$, the expected length of the search path for x is $H_{\lceil x \rceil} + H_{n - \lceil x \rceil}$.*

Random Binary Trees (Proof 1/2)

Let I_i be the indicator random variable that is equal to one when i appears on the search path for x and zero otherwise. The length of the search path is given by

$$\sum_{i \in \{0,\ldots,n-1\} \setminus \{x\}} I_i$$

If $x \in \{0, \ldots, n-1\}$, the expected length of the search path is given by

Random Binary Trees (Proof 2/2)

$$
\begin{aligned}
\mathrm{E}\left[\sum_{i=0}^{x-1} I_i + \sum_{i=x+1}^{n-1} I_i\right] &= \sum_{i=0}^{x-1} \mathrm{E}\left[I_i\right] + \sum_{i=x+1}^{n-1} \mathrm{E}\left[I_i\right] \\
&= \sum_{i=0}^{x-1} 1/(\lfloor x \rfloor - i + 1) + \sum_{i=x+1}^{n-1} 1/(i - \lceil x \rceil + 1) \\
&= \sum_{i=0}^{x-1} 1/(x - i + 1) + \sum_{i=x+1}^{n-1} 1/(i - x + 1) \\
&= \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{x+1} \\
&\quad + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-x} \\
&= H_{x+1} + H_{n-x} - 2 \enspace .
\end{aligned}
$$

The corresponding calculations for a search value $x \in (-1, n) \setminus \{0, \ldots, n-1\}$ are almost identical

Summary

Theorem

A random binary search tree can be constructed in $O(n \log n)$ *time. In a random binary search tree, the* $\operatorname{find}(x)$ *operation takes* $O(\log n)$ *expected time.*

- In particular a random binary tree implements a sorted dictionary where $find(x)$ $add(x, v)$ and $remove(x)$ take $O(\log n)$ expected time.