ADT for **Quick Search**

*Tree* data structure

Lecture 17

*by Marina Barsky*
Motivation 1: Searching

Find company address in the address book
Motivation 2: Closest Height

Find people in your class whose height is closest to yours.
Motivation 3: Date Ranges

Find all emails received in a given period

<table>
<thead>
<tr>
<th>FROM</th>
<th>KNOW</th>
<th>TO</th>
<th>SUBJECT</th>
<th>SENT TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;lawiki.i2p admin&quot;</td>
<td></td>
<td>Bote User &lt;uhOd&gt;</td>
<td>hi</td>
<td>Unknown</td>
</tr>
<tr>
<td>anonymous</td>
<td></td>
<td>Bote User &lt;uhOd&gt;</td>
<td>Sanders 2016</td>
<td>Aug 30, 2015</td>
</tr>
<tr>
<td>anonymous</td>
<td></td>
<td>Bote User &lt;uhOd&gt;</td>
<td>I2PCon 2016</td>
<td>Aug 30, 2015</td>
</tr>
<tr>
<td>Anon Developer &lt;gvbM&gt;</td>
<td></td>
<td>Bote User &lt;uhOd&gt;</td>
<td>Re: Bote changess</td>
<td>Aug 30, 2015</td>
</tr>
<tr>
<td>I2P User &lt;uUXx&gt;</td>
<td></td>
<td>Bote User &lt;uhOd&gt;</td>
<td>Hello World!</td>
<td>Aug 30, 2015</td>
</tr>
</tbody>
</table>
Motivation 4: Partial Search

Find all words that **start with** some given *prefix*
**Specification**

A *Quick Search ADT* stores a number of elements each with a *key* and supports the following operations:

→ **Search**(x): returns the element with the key=x

→ **Range**(lo, hi): returns all elements with keys between lo and hi

→ **NearestNeighbor**(x): returns an element with the key closest to x
Range(5, 13)

Search(7)

NearestNeighbor(5)
Is seems that the best idea is to store the elements *sorted* by keys
How to make this dynamic?

- Store keys in **sorted order**
- But we also want to be able to add/remove keys efficiently
A **Quick Search ADT** stores a number of elements each with a *key* and supports the following operations:

- **Search**\( (x) \): returns the element with the key=\( x \)
- **Range**\( (lo, hi) \): returns all elements with keys between \( lo \) and \( hi \)
- **NearestNeighbor**\( (x) \): returns an element with the key closest to \( x \)
- **Insert**\( (x) \): adds an element with key \( x \)
- **Remove**\( (x) \): removes the element with key \( x \)
Example

Insert (3)

Remove (10)
## Possible Implementations

Let’s try known data structures:

- Array
- Sorted array
- Linked list
Array

→ Range Search: $O(n) \times$

range(1, 7)

7 10 4 13 1 6 15
Array

- **Range Search:** $O(n) \times$
- **Nearest Neighbor:** $O(n) \times$

```plaintext
nearestNeighbor(6)
```

```
| 7 | 10 | 4 | 13 | 1 | 6 | 15 |
```
Array

- Range Search: \(O(n)\) ×
- Nearest Neighbor: \(O(n)\) ×
- Insert: \(O(1)\) ✓

<table>
<thead>
<tr>
<th>7</th>
<th>10</th>
<th>4</th>
<th>13</th>
<th>1</th>
<th>6</th>
<th>15</th>
</tr>
</thead>
</table>

insert (3)

3
Array

- Range Search: $O(n)$ ×
- Nearest Neighbor: $O(n)$ ×
- Insert: $O(1)$ ✓
- Remove: $O(1)^* ✓$

delete (10)

swap

The order of elements does not matter!

After locating an index of the element to be removed
Sorted Array

→ Range Search: $O(\log(n))$ Yes
Sorted Array

- Range Search: $O(\log(n)) \checkmark$
- Nearest Neighbor: $O(\log(n)) \checkmark$

nearestNeighbor(3)

1 3 4 7 10 13 15
Sorted Array

- Range Search: $O(\log(n))$  
- Nearest Neighbor: $O(\log(n))$  
- Insert: $O(n)$  

Insert (6) must keep in sorted order – so shift

```
1  3  4  6  7  10  13  15
```
Sorted Array

- Range Search: \( O(\log(n)) \) ✔️
- Nearest Neighbor: \( O(\log(n)) \) ✔️
- Insert: \( O(n) \) ✗
- Remove: \( O(n) \) ✗

Cannot have gaps – shift again

delete (6)
Linked List

→ Range Search: $O(n) \times$

range (4, 9)
Linked List

→ Range Search: $O(n) \times$
→ Nearest Neighbor: $O(n) \times$

nearestNeighbor(13)
Linked List

- Range Search: $O(n) \times$
- Nearest Neighbor: $O(n) \times$
- Insert: $O(1) \checkmark$

insert (3)
Linked List

- **Range Search:** $O(n)$ ✗
- **Nearest Neighbor:** $O(n)$ ✗
- **Insert:** $O(1)$ ✓
- **Remove:** $O(1)^*$ ✓

*after locating the node with the element to be removed*
Nothing works!

- We want an efficient data structure for fast search and update operations
- None of the known data structures work
- Sorted arrays are good for search but not for update

We need something new...
Recall: Binary Search
What if we record search questions...
We will get a tree

Binary Search Tree
New Data Structure: *Tree*

Natalie Jeremijenko, *Tree Logic*, Massachusetts Museum of Contemporary Art (MASS MoCA), 1999
Biology:
Phylogenetic Tree of animals
Natural Language Processing: Syntax Tree

S
  NP
    N
    He
  VP
    V
    looked
  PP
    at
    D
    the
    N
    dog
    PP
      with
      D
      one
      N
      eye
Computer programs: Expression Tree

3 + ((5 + 9) * 2)
Quick Search: 
Binary Search Tree
Tree - new recursive data structure

- Main element of the tree: node
- Each node contains data and an array of links to the child nodes

```python
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.children = []
        self.parent = None
```

```python
class TreeNode {
    int data;
    TreeNode [] children;
    [TreeNode parent;]
}
```
Tree is defined by a single reference variable $root$

$Tree$ is either

- Null (empty tree)
- Root node which contains data and links to child nodes
Binary tree: each node has 2 children

```
class Node {
    int data;
    Node left;
    Node right;
    [Node parent;]
}
```

Either Left or Right can be null (empty tree)
Tree terminology: parent and child
Tree terminology: parent and child

Have direct relationship
Tree terminology: node and edge

An edge connects nodes: parent-child or child-parent relationships
Tree terminology: **root**

*The parent of all nodes, the starting point*
Tree terminology: *ancestor* and *descendant*.

Ancestor: parent, or parent of parent, etc.
Tree terminology: ancestor and descendant

Descendant: child, or child of child, etc.
Tree terminology: siblings

Sharing the same parent
Tree terminology: leaves and interior (internal) nodes

In a leaf node both children are empty trees
Tree levels and node depth

Level 0
 depth = 0

Level 1
 depth = 1

Level 2
 depth = 2

Level 3
 depth = 3

Distance from the root: how many edges to go from the root to the node
Node height

Level 0
depth = 0

Level 1
depth = 1

Level 2
depth = 2

Level 3
depth = 3

Distance from the node to the bottom: how many edges to go to the furthest leaf
Algorithm `height (node)`

```python
if node == null:
    return 0
if node.left == null and node.right == null:
    return 0
return 1 + Max(height(node.left), height(node.right))
```
Algorithm \textit{size} (\textit{tree})

\begin{verbatim}
if \textit{tree} == \textit{null}
    return 0
return 1 + \textit{size} (\textit{tree}.left) + \textit{size} (\textit{tree}.right)
\end{verbatim}

Recursive algorithms are common
Which of the following correctly computes the *depth* of a given tree node in the non-empty tree?

A. Algorithm depth(node)
   
   ```java
   if node == null
       return 0
   return 1 + depth(node.parent)
   ```

B. Algorithm depth(node)
   
   ```java
   if node.parent == null
       return 0
   return 1 + depth(node.parent)
   ```

C. Algorithm depth(node)
   
   ```java
   if node == null
       return -1
   return 1 + depth(node.parent)
   ```

D. More than one is correct

E. None is correct
Tree traversals

- Task: list all the nodes in the tree

Two types of traversals:

- **Depth-first**: we completely traverse one sub-tree before exploring a sibling sub-tree
- **Breadth-first**: We traverse all nodes at one level before progressing to the next level
Depth-first tree traversals

- In-order
- Pre-order
- Post-order
Depth-first: in order

Algorithm \textbf{InOrderTraversal}(tree)

\begin{enumerate}
\item \textbf{if} $\text{tree} == \text{Null}$ :
  \begin{enumerate}
  \item \textbf{return}
  \item \textbf{InOrderTraversal}(tree.left)
  \item \textbf{print} (tree.key)
  \item \textbf{InOrderTraversal}(tree.right)
  \end{enumerate}
\end{enumerate}
Which sequence of nodes is obtained as a result of **in-order traversal** of the tree on the left?

A. abdhiecfg
B. hdibeafcg
C. ahdibefgc
D. More than one is correct
E. None is correct
In-order
In-order
In-order

A B

A B C D E F G
In-order

A B C D B A C F E G
In-order
In-order

A B C D
In-order

A B C D E

A B C D E
In-order

A B C D E F
In-order
In-order

A B C D E F G

node D

left subtree of D

right subtree of D
Depth-first: pre-order

Algorithm `PreOrderTraversal(tree)`

```python
if tree == null:
    return
print (tree.key)
PreOrderTraversal(tree.left)
PreOrderTraversal(tree.right)
```

me first

me → left → right
Which sequence of nodes is obtained as a result of pre-order traversal of the tree on the left?

A. abdhiecfg
B. abcdehcfg
C. abdhiecfg
D. More than one is correct
E. None is correct
Pre-order
Pre-order

D B
Pre-order

D B A

A

B

C

E

F

G

D

Pre-order

D B A C F
Pre-order

D B A C F E
Pre-order

D B A C F E G
Pre-order

me, node D

left subtree of D right subtree of D
Algorithm **PostOrderTraversal**(tree)

```python
if tree == null:
    return

PostOrderTraversal(tree.left)
PostOrderTraversal(tree.right)
print(tree.key)
```

**Depth-first: post-order**

- Children first
- left → right → me
Which sequence of nodes is obtained as a result of post-order traversal of the tree on the left?

A. abdhiecfg
B. abcdehifg
C. hidebfgca
D. More than one is correct
E. None is correct
Post-order
Post-order
Post-order
Post-order
Post-order

A B C D E F G
Post-order

A C B
Post-order

A C B
Post-order

A C B E
Post-order

A C B E G
Post-order

A C B E G F
Post-order

A C B E G F D
Post-order

A C B  E G F  D

left subtree of D  right subtree of D

me, node D
Breadth-first traversal

Level traversal:  
D  
B F  
A C E G
Algorithm \textit{BreadthFirstTraversal}(tree)

if \( \text{tree} == \text{null} \):
    return

Queue \textit{q}

\textit{q}.enqueue(\textit{tree})

while not \textit{q}.is\textit{Empty}():
    node ← \textit{q}.dequeue()
    print(node)
    if node.left != null:
        \textit{q}.enqueue(node.left)
    if node.right != null:
        \textit{q}.enqueue(node.right)
Breadth first: level traversal

Queue: D

Output:
Breadth first: level traversal

Queue:
Output: D
Breadth first: level traversal

Queue: B F
Output: D
Breadth first: level traversal

Queue: B F
Output: D
Breadth first: level traversal

Queue: F
Output: D B
Breadth first: level traversal

Queue: F A C
Output: D B
Breadth first: level traversal

Queue: F A C
Output: D B
Breadth first: level traversal

Queue: A C
Output: D B F
Breadth first: level traversal

Queue: A C E G
Output: D B F
Breadth first: level traversal

Queue: A C E G
Output: D B F
Breadth first: level traversal

Queue: C E G

Output: D B F A
Breadth first: level traversal

Queue: C E G
Output: D B F A
Breadth first: level traversal

Queue: E G
Output: D B F A C
Breadth first: level traversal

Queue: E G
Output: D B F A C
Breadth first: level traversal

Queue: G
Output: D B F A C E
Breadth first: level traversal

Queue: G
Output: D B F A C E
Breadth first: level traversal

Queue: empty
Output: D B F A C E G
Tree data structure: notes

➢ Tree is fully defined by its root node
➢ Each node has (at least) a key and links to children
➢ Tree traversals:
  ○ Depth-first: uses recursion (stack)
    ■ pre-order
    ■ in-order
    ■ post-order
  ○ Breadth-first: uses queue
➢ When working with a tree, recursive algorithms are common
➢ In Computer Science, trees grow down!
Can you guess which (real) words are spelled by some type of traversal of these trees?
Which type of traversal is used to spell these words?

A. in-order, pre-order, post-order
B. pre-order, post-order, in-order
C. post-order, in-order, pre-order
D. None of the above (something else)
For completeness: **breadth-first** traversal
For completeness: **breadth-first** traversal