Binary Search Trees: Introduction

Data Structures

Data Structures and Algorithms

Learning Objectives

- Provide examples of the sorts of problems we hope to solve with Binary Search Trees.
- Show why data structures that we have already covered are insufficient.

Outline

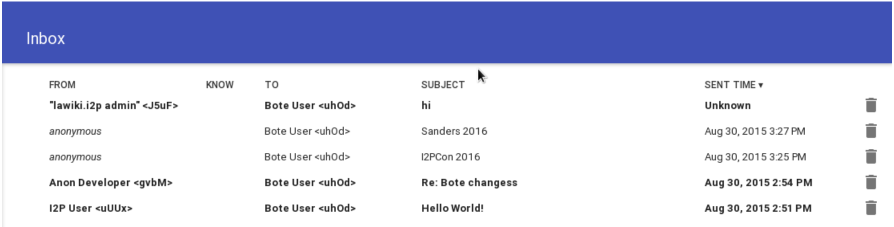
1. Local Search
2. Attempts

Dictionary Search

Find all words that start with some given string.

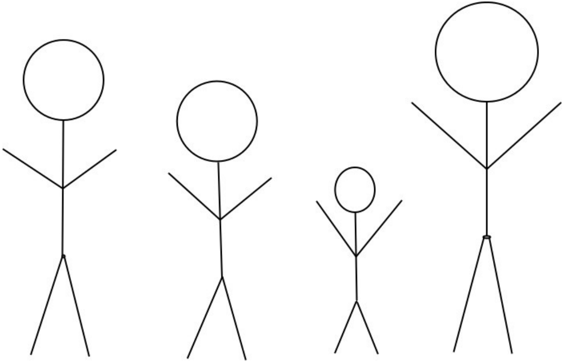
Date Ranges

Find all emails received in a given period.



Closest Height

Find the person in your class whose height is closest to yours.



Local Search

Definition

A Local Search Datastructure stores a number of elements each with a key coming from an ordered set. It supports operations:

- `RangeSearch`(x, y): Returns all elements with keys between x and y.
- `NearestNeighbors`(z): Returns the element with keys on either side of z.

Example

1	4	6	7	10	13	15

Example

1	4	6	7	10	13	15

`RangeSearch`(5, 12)

1	4	6	7	10	13	15

Example

1	4	6	7	10	13	15

`RangeSearch`(5, 12)

1	4	6	7	10	13	15

`NearestNeighbors`(3)

1	4	6	7	10	13	15

Dynamic Data Structure

We would also like to be able to modify the data structure as we go.

- `Insert`(*x*): Adds a element with key *x*.
- `Delete`(*x*): Removes the element with key *x*.

Example

1	4	6	7	10	13	15

Example

1	4	6	7	10	13	15

`Insert`(3)

1	3	4	6	7	10	13	15

Example

1	4	6	7	10	13	15

`Insert`(3)

1	3	4	6	7	10	13	15

`Delete`(10)

1	3	4	6	7	13	15

Problem

If an empty data structure is given these commands what does it output at the end?

- `Insert`(3)
- `Insert`(8)
- `Insert`(5)
- `Insert`(10)
- `Delete`(8)
- `Insert`(12)
- `NearestNeighbors`(7)

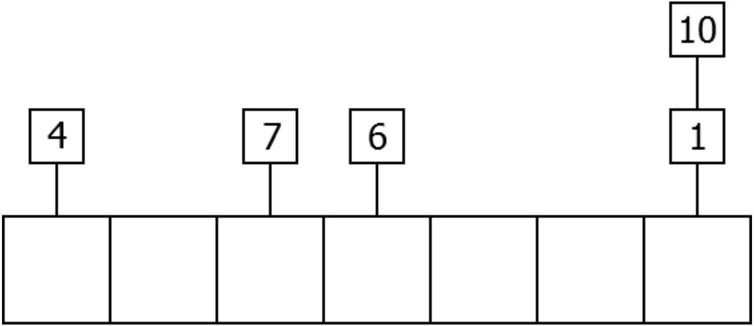
Answer

3	5	~~8~~	10	12

Outline

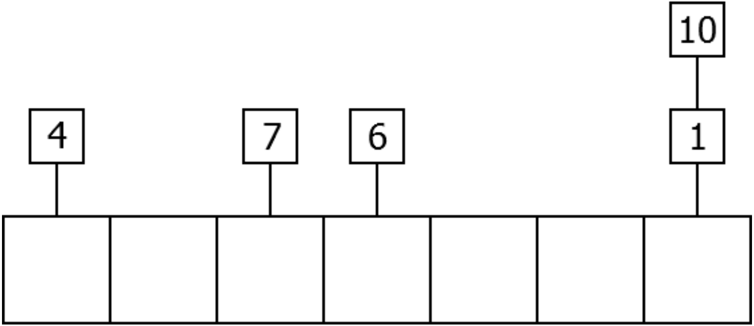
Hash Table

- `RangeSearch`: Impossible ×



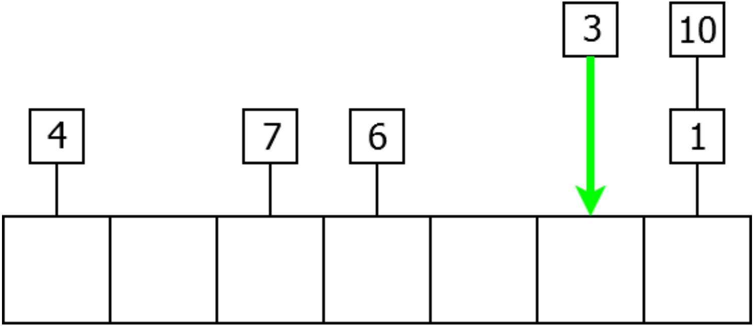
Hash Table

- `RangeSearch`: Impossible ×
- `NearestNeighbors`: Impossible ×



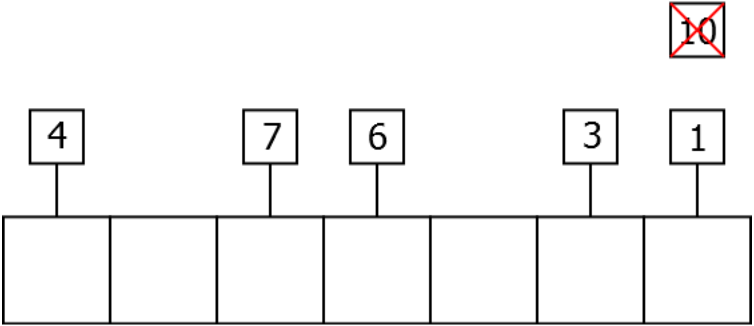
Hash Table

- `RangeSearch`: Impossible ✗
- `NearestNeighbors`: Impossible ✗
- `Insert`: $O(1)$ ✓



Hash Table

- `RangeSearch`: Impossible ×
- `NearestNeighbors`: Impossible ×
- `Insert`: $O(1)$ ✓
- `Delete`: $O(1)$ ✓



Array

- RangeSearch: $O(n)$ ×

7	10	4	13	1	6	15

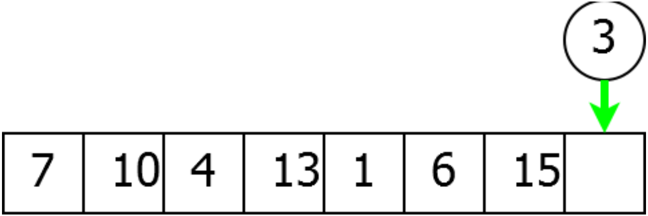
Array

- `RangeSearch`: $O(n)$ ×
- `NearestNeighbors`: $O(n)$ ×

7	10	4	13	1	6	15

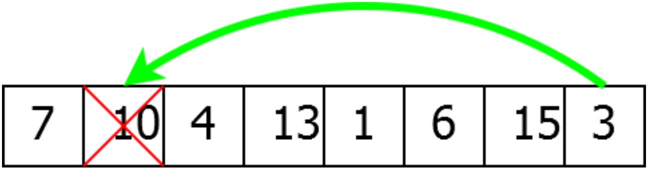
Array

- `RangeSearch`: $O(n)$ ✗
- `NearestNeighbors`: $O(n)$ ✗
- `Insert`: $O(1)$ ✓



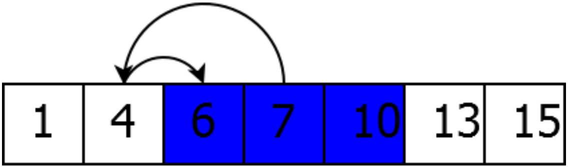
Array

- `RangeSearch`: $O(n)$ ×
- `NearestNeighbors`: $O(n)$ ×
- `Insert`: $O(1)$ ✓
- `Delete`: $O(1)$ ✓



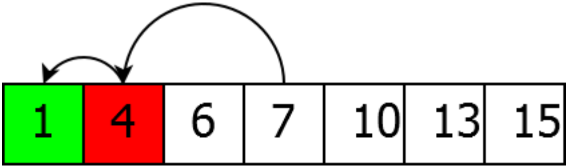
Sorted Array

- `RangeSearch`: $O(\log(n))$ ✓



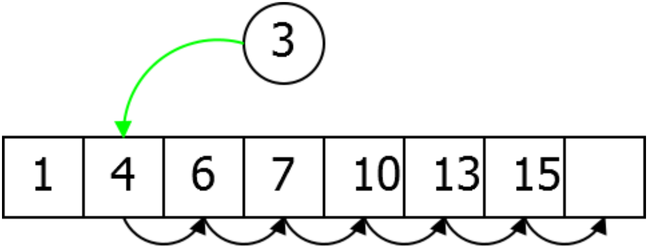
Sorted Array

- `RangeSearch`: $O(\log(n))$ ✓
- `NearestNeighbors`: $O(\log(n))$ ✓



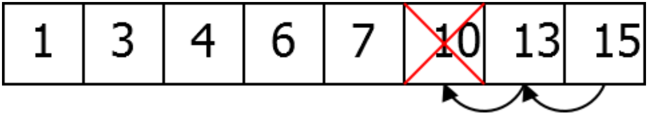
Sorted Array

- `RangeSearch`: $O(\log(n))$ ✓
- `NearestNeighbors`: $O(\log(n))$ ✓
- `Insert`: $O(n)$ ×



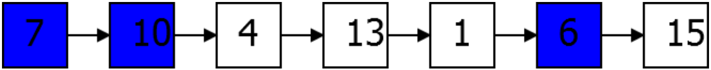
Sorted Array

- `RangeSearch`: $O(\log(n))$ ✓
- `NearestNeighbors`: $O(\log(n))$ ✓
- `Insert`: $O(n)$ ×
- `Delete`: $O(n)$ ×



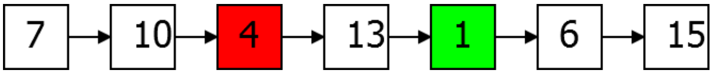
Linked List

- `RangeSearch`: $O(n)$ ×



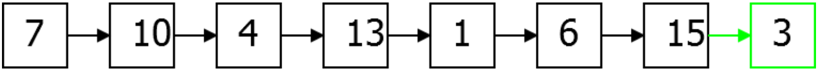
Linked List

- `RangeSearch`: $O(n)$ ×
- `NearestNeighbors`: $O(n)$ ×



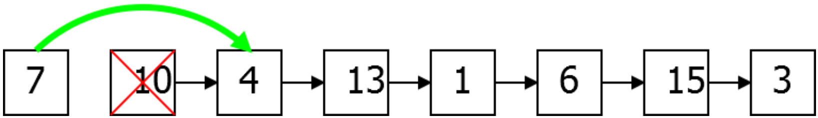
Linked List

- `RangeSearch`: $O(n)$ ×
- `NearestNeighbors`: $O(n)$ ×
- `Insert`: $O(1)$ ✓



Linked List

- `RangeSearch`: $O(n)$ ×
- `NearestNeighbors`: $O(n)$ ×
- `Insert`: $O(1)$ ✓
- `Delete`: $O(1)$ ✓



Need Something New

Problem

Previous data structures won’t work. We need something new.

Binary Search Trees: Search Trees

Learning Objectives

- Describe how a Binary Search Tree data structure is constructed.
- Determine whether a tree is properly sorted.

Last Time

- Want data structure for local search.

Last Time

- Want data structure for local search.
- None of the existing data structures work.

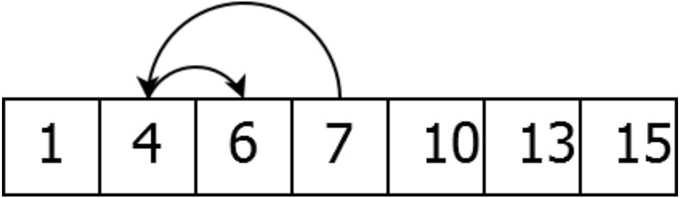
Last Time

- Want data structure for local search.
- None of the existing data structures work.
- Sorted arrays can search but not update.

Outline

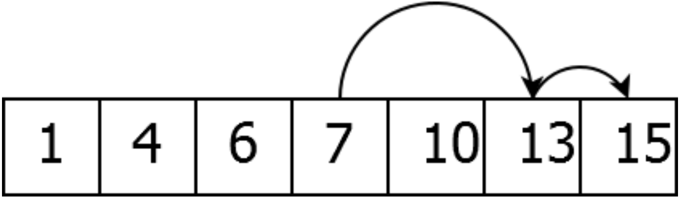
Binary Search

Search an array:



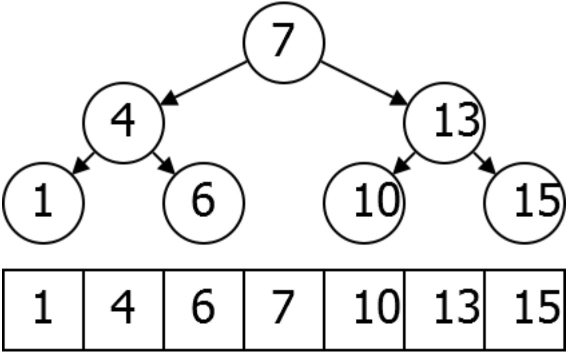
Binary Search

Search an array:



Search Tree

Consider questions asked:



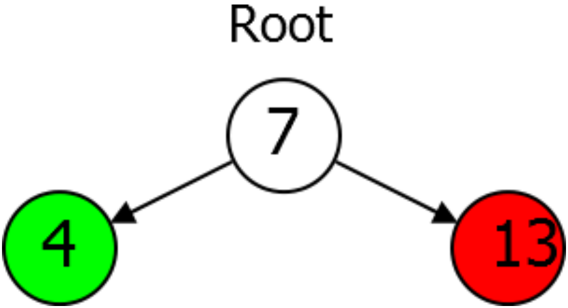
The search tree is much easier to insert into.

Outline

1. Array Search
2. The Search Tree Structure

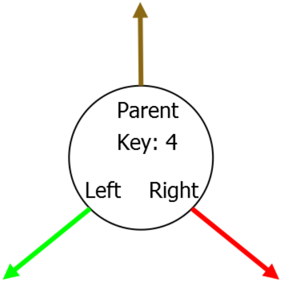
Parts of a Tree

- Root node.
- `Left` subtree smaller keys.
- `Right` subtree bigger keys.



Tree Node Data Type

- `Key`
- `Parent`
- `Left` Child
- `Right` Child

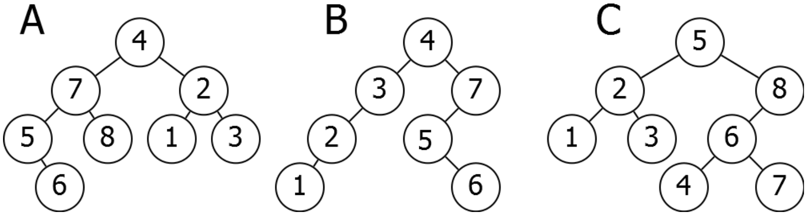


Search Tree Property

X's key is larger than the key of any descendent of its left child, and smaller than the key of any descendant of its right child.

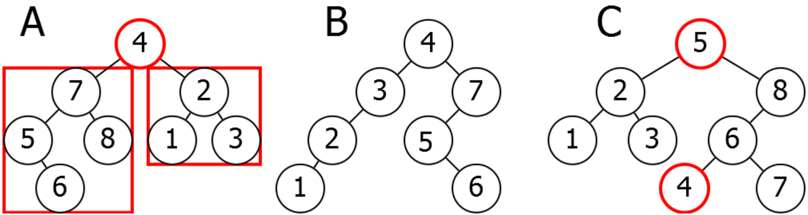
Problem

Which of the following Trees satisfies the Search Tree Property?



Problem

Which of the following Trees satisfies the Search Tree Property?



Next Time

How to do basic operations on Binary Search Trees.