05 A: Trees II

CS1102S: Data Structures and Algorithms

Martin Henz

February 10, 2010

Generated on Wednesday 10th February, 2010, 10:54

Motivation

Trees in computer science

Trees are ubiquitous in CS, covering operating systems, computer graphics, data bases, etc.

Motivation

Trees in computer science

Trees are ubiquitous in CS, covering operating systems, computer graphics, data bases, etc.

Trees as data structures

Provide $O(\log N)$ search operations

Motivation

Trees in computer science

Trees are ubiquitous in CS, covering operating systems, computer graphics, data bases, etc.

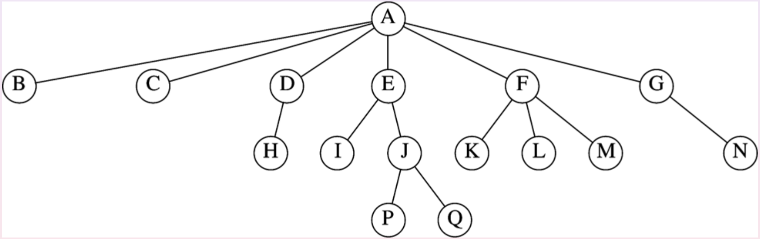
Trees as data structures

Provide $O(\log N)$ search operations

Heaps

Serve as basis for other efficient data structures, such as heaps

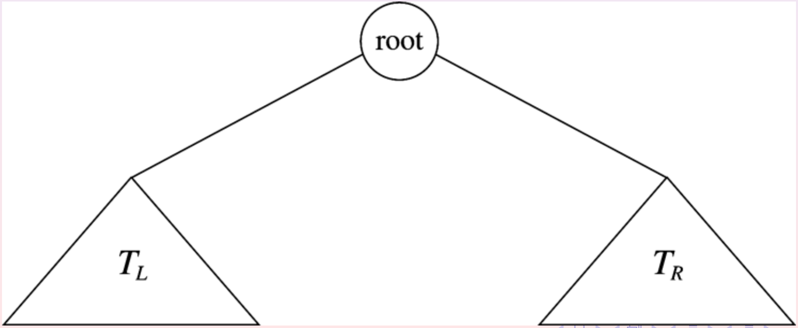
Example



Binary Trees

Definition

A binary tree is a tree in which no node can have more than two children.



Implementation

```
ClassBinaryNode {
  // accessible by other package routines
  Object      element;     // The data in the node
  BinaryNode  left;        // Left child
  BinaryNode  right;       // Right child
}
```

Motivation

Setup

We would like to quickly find out if a given data item is included in a collection.

Motivation

Setup

We would like to quickly find out if a given data item is included in a collection.

Example

In an underground carpark, a system captures the licence plate numbers of incoming and outgoing cars.

Motivation

Setup

We would like to quickly find out if a given data item is included in a collection.

Example

In an underground carpark, a system captures the licence plate numbers of incoming and outgoing cars.
Problem: Find out if a particular car is in the carpark.

Operations for Sets

```
interface Set<T> {
    public void add(T x);
    // same as insert(T x);

    public void remove(T x);
    public boolean contains(T x);
    ...
}
```

How About Lists, Arrays, Stacks, Queues?

Problem with Lists, Arrays, Stacks, Queues

With lists, arrays, stacks and queues, we can only access the collection using an index or in a LIFO/FIFO manner.

How About Lists, Arrays, Stacks, Queues?

Problem with Lists, Arrays, Stacks, Queues

With lists, arrays, stacks and queues, we can only access the collection using an index or in a LIFO/FIFO manner. Therefore, search takes linear time.

How About Lists, Arrays, Stacks, Queues?

Problem with Lists, Arrays, Stacks, Queues

With lists, arrays, stacks and queues, we can only access the collection using an index or in a LIFO/FIFO manner. Therefore, search takes linear time.

How to avoid linear access?

For efficient data structures, we often exploit properties of data items.

Example

Simple license plates

Let us say the license plate numbers are positive integers from 0 to 9999.

Example

Simple license plates

Let us say the license plate numbers are positive integers from 0 to 9999.

Solution

- Keep an array inCarPark of boolean values (initially all false).

Example

Simple license plates

Let us say the license plate numbers are positive integers from 0 to 9999.

Solution

- Keep an array inCarPark of boolean values (initially all false).
- insert (i) sets inCarPark[i] to true

Example

Simple license plates

Let us say the license plate numbers are positive integers from 0 to 9999.

Solution

- Keep an array inCarPark of boolean values (initially all false).
- insert (i) sets inCarPark[i] to true
- remove(i) sets inCarPark[i] to false

Example

Simple license plates

Let us say the license plate numbers are positive integers from 0 to 9999.

Solution

- Keep an array inCarPark of boolean values (initially all false).
- insert(i) sets inCarPark[i] to true
- remove(i) sets inCarPark[i] to false
- contains(i) returns inCarPark[i].

The Sad Truth

Not all data items are small integers!

In Singapore, license plate numbers start with 2–3 letters, followed by a number, followed by another letter.

The Sad Truth

Not all data items are small integers!

In Singapore, license plate numbers start with 2–3 letters, followed by a number, followed by another letter.

But: one property remains

We can *compare* two license plate numbers, for example lexicographically.

Lexicographic Ordering on License Plate Numbers

- First compare the first letters as in a dictionary e.g. “SBX...” $<$ “SCY...”, “SA...” $<$ “SAB...”

Lexicographic Ordering on License Plate Numbers

- First compare the first letters as in a dictionary e.g. “SBX...” $<$ “SCY...”, “SA...” $<$ “SAB...”
- If the letters are the same, use the following number e.g. “SBX 100” $<$ “SBX 101”

Lexicographic Ordering on License Plate Numbers

- First compare the first letters as in a dictionary e.g. "SBX..." $<$ "SCY...", "SA..." $<$ "SAB..."
- If the letters are the same, use the following number e.g. "SBX 100" $<$ "SBX 101"
- If the letters and numbers are the same, use the final letter e.g. "SBX 101 P" $<$ "SBX 101 Q"

The Comparable Interface

API Interface Comparable

```
interface Comparable<T> {
   public int compareTo(T o);
}
```

Mathematics of Comparable

Ordering

Instances of the Comparable interface are subject to a *total ordering*. For any two elements x and y, we know whether:

- x smaller then y: x.compareTo(y) returns negative int
- x smaller then y: x.compareTo(y) returns positive int
- x equals y: x.compareTo(y) returns 0

Excursion: Bounded Types

Type variables

allow the programmer to refer to a type at multiple places.

Example

```
public  static <Any> SchemeList<Any>
    concatAll(SchemeList<SchemeList<Any>>
              aListList                  ) {
    ...
}
```

Excursion: Bounded Types

Wildcard Types

Sometimes, a generic type is completely unrestricted. We use ? without having to declare it.

Example

```
public static int
  iterativeLength(SchemeList<?> aList) {
  int acc = 0;
  while (! aList.isNil()) {
    aList = aList.cdr();
    acc++; }
  return acc; }
```

Excursion: Bounded Types

Upper bounds for types

Sometimes, a type variable must be *bounded* to restrict the types that it stands for to a class and all its sub-classes.

Example

```
interface Collection<E> { ...
  boolean add(E e);
  boolean addAll(Collection<? extends E> c);
  ...
}
```

Excursion: Bounded Types

interface Comparable

```
interface Comparable<T> {
    public int compareTo(T o);
}
```

Invariance of generic types

If Lion is a subtype of Animal, then Cage<Lion> is *not* a subtype of Cage<Animal>.

Excursion: Bounded Types

Invariance of generic types

If Lion is a subtype of Animal, then Cage$<$Lion$>$ is *not* a subtype of Cage$<$Animal$>$.

Excursion: Bounded Types

Invariance of generic types

If Lion is a subtype of Animal, then Cage<Lion> is *not* a subtype of Cage<Animal>.

Invariance of Comparable

Therefore, if Animal implements Comparable<Animal>, Lion does *not necessarily* implement Comparable<Lion>.

Excursion: Bounded Types

Invariance of generic types

If Lion is a subtype of Animal, then Cage<Lion> is *not* a subtype of Cage<Animal>.

Invariance of Comparable

Therefore, if Animal implements Comparable<Animal>, Lion does *not necessarily* implement Comparable<Lion>.

Lower bounds for Comparable

We want to allow Lion to implement Comparable<T> as long as T is a super type of Lion.

Excursion: Bounded Types

Lower bounds for Comparable

We want to allow Lion to implement Comparable<T> as long as T is a super type of Lion.

```
class BinarySearchTree
      <Any extends Comparable<? super Any>>
   {...}
```

Binary Search

Setup

Keep items in a tree. Each node holds one data item.

Binary Search

Setup

Keep items in a tree. Each node holds one data item.

Idea

The left subtree of a node V only contains items smaller than V and the right subtree only contains items larger than V.

Binary Search

Setup

Keep items in a tree. Each node holds one data item.

Idea

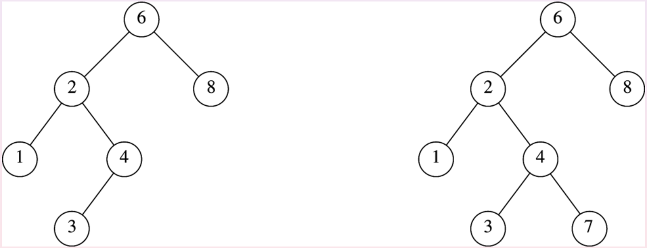
The left subtree of a node V only contains items smaller than V and the right subtree only contains items larger than V.

Search

can then proceed top-down, starting at the root. If the search item is smaller than the item at the root, go down to the left, and if it is larger, go right.

Example

Both trees are binary trees, but only the left tree is a search tree.



Implementation

```
private static class BinaryNode<AnyType>{
  AnyType element;
  BinaryNode<AnyType> left;
  BinaryNode<AnyType> right;
  BinaryNode( AnyType theElement ) {
    this( theElement, null, null ); }
  BinaryNode(AnyType theElement,
             BinaryNode<AnyType> lt,
             BinaryNode<AnyType> rt) {
    element = theElement;
    left = lt; right = rt; }
}
```

Implementation

```
public class
BinarySearchTree<AnyType extends
                Comparable<? super AnyType>> {
  private static class BinaryNode<AnyType> {..}
  private BinaryNode<AnyType> root;
  public BinarySearchTree() {
    root = null; }
  public void makeEmpty() {
    root = null; }
  public boolean isEmpty() {
    return root == null; }
  ...
}
```

Implementation

```
public class
BinarySearchTree<AnyType extends
                 Comparable<? super AnyType>> {
  ...
  public boolean contains( AnyType x ) {
    return contains( x, root ); }
  public AnyType findMin( ) { // findMax similar
    if( isEmpty( ) ) throw new UnderflowException(
    return findMin( root ).element; }
  public void insert( AnyType x ) {
    root = insert( x, root ); }
  public void remove( AnyType x ) {
    root = remove( x, root ); }
```

Implementation of Search

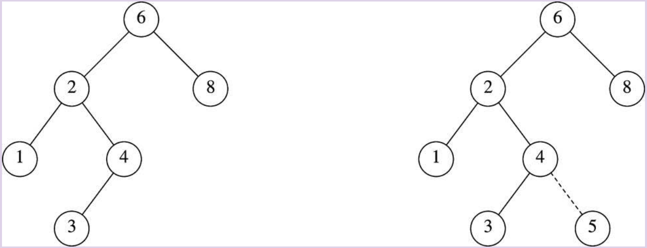
```
private boolean contains(AnyType x,
                         BinaryNode<AnyType> t ) {
  if( t == null ) return false;
  int compareResult = x.compareTo( t.element );
  if (compareResult < 0)
    return contains( x, t.left );
  else if( compareResult > 0 )
    return contains( x, t.right );
  else
    return true; }
```

Insertion

Idea

Proceed like in search. If item is found, do nothing. If not, insert it in the last visited position.

Example



Deletion

Idea

Proceed like in search. If item is not found, do nothing. If item is found, take action depending on node.

Deletion

Idea

Proceed like in search. If item is not found, do nothing. If item is found, take action depending on node.

Leaf

If the node is leaf, delete it from parent.

Deletion

Idea

Proceed like in search. If item is not found, do nothing. If item is found, take action depending on node.

Leaf

If the node is leaf, delete it from parent.

One child

If the node has one child, move the child to parent.

Example: Deletion of Node with One Child

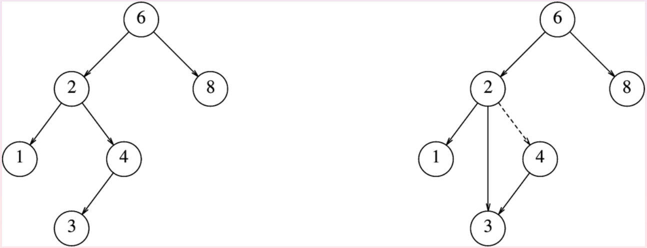
One child

If the node has one child, move the child to parent.

Example: Deletion of Node with One Child

One child

If the node has one child, move the child to parent.



Deletion of Node with Two Children

Idea

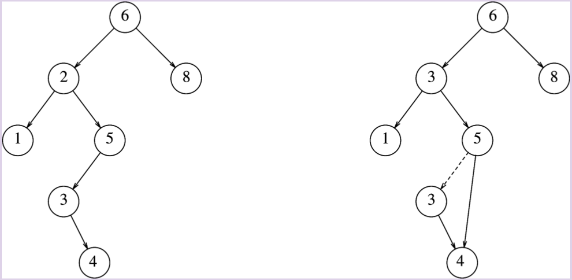
Replace data with data of smallest child on the right; then delete smallest child on the right.

Deletion of Node with Two Children

Idea

Replace data with data of smallest child on the right; then delete smallest child on the right.

Example



Average-case Analysis

Average Depth

If all insertion sequences are equally likely, the average depth of any node is $O(\log N)$ (proof in Chapter 7)

Average-case Analysis

Average Depth

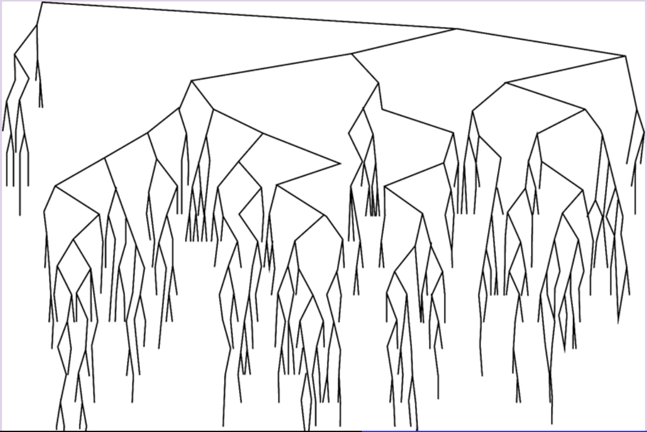
If all insertion sequences are equally likely, the average depth of any node is $O(\log N)$ (proof in Chapter 7)

Deletion introduces imbalance

Deletion favours right subtree, and therefore trees become "left-heavy" on the long run.

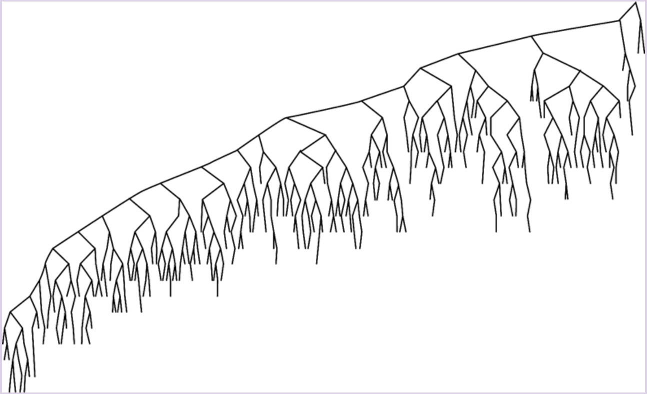
Average-case Analysis

Randomly generated binary search tree



Average-case Analysis

Search tree after N^2 insert/delete



1. Review: Trees
2. Binary Search Trees
3. Sets in Java Collections API

Sets

Idea

A Set (interface) is a Collection (interface) that does not allow duplicate entries.

Sorted Sets

A SortedSet (interface) assumes that the data items are comparable (using a Comparator operation).

```
interface SortedSet<E> extends Set<E>
```

Implementation

The most common implementation of SortedSet is TreeSet.

Next Week

- Friday: Midterm
- Monday Lab: Lab tasks (attendance taken)
- Wednesday: Hashing
- Thursday: Tutorial on midterm solutions
- Friday: Priority Queues