Algorithm Engineering Lecture 4: More Data Structures for Your Problem-Solving Toolkit

CSE 431

Spring 2024

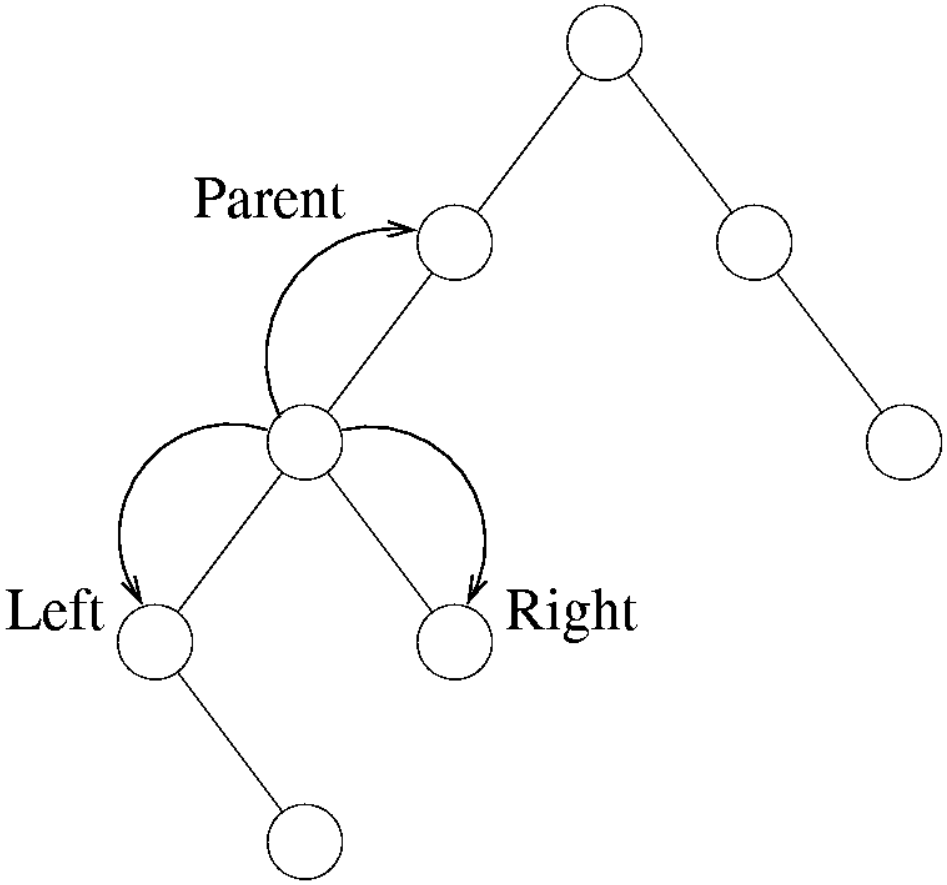
Kevin Liu, Ph.D.

Michigan State University

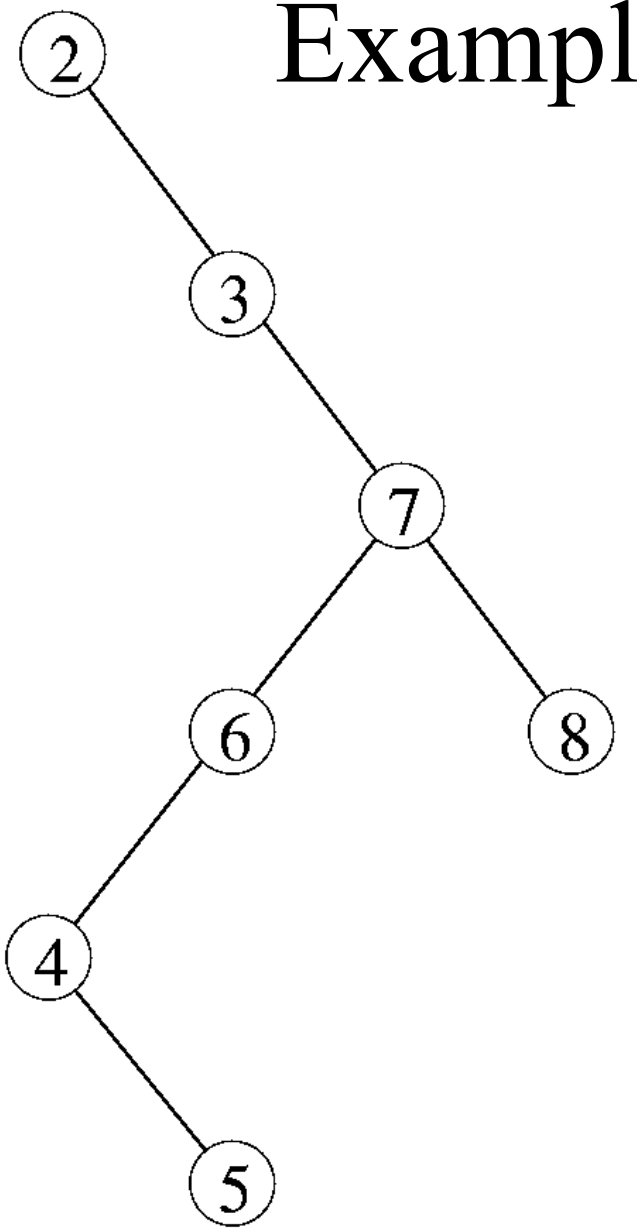
Optional reading:
Skiena 2012
chapter 3

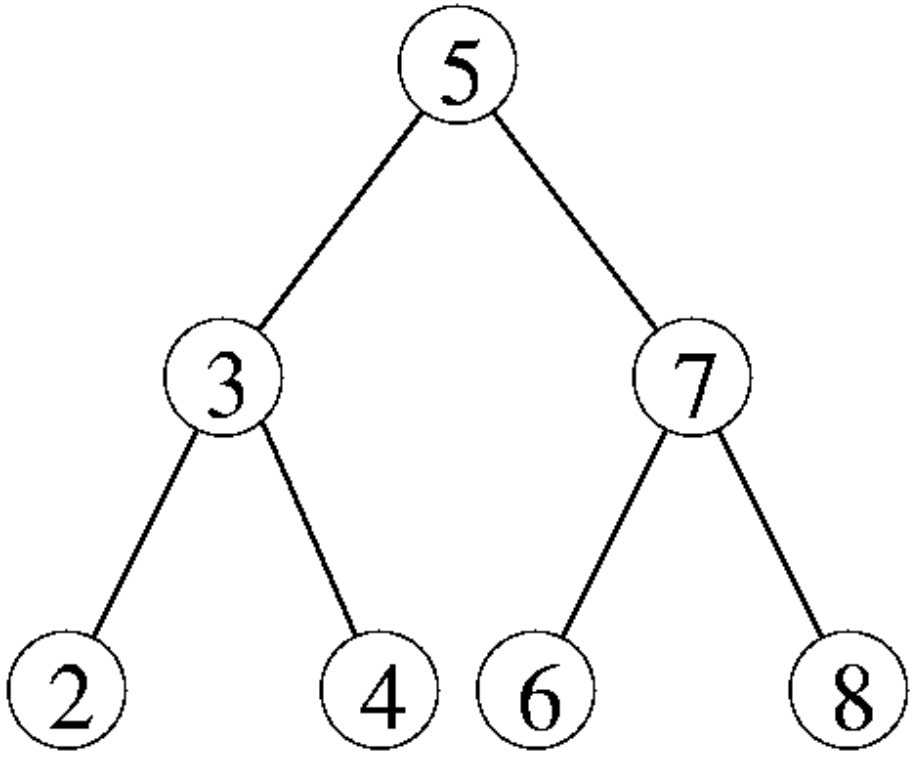
Includes material adapted from
Stony Brook 373 and
previous editions of CSE431

Binary Trees



Example Search Trees





Successor and Predecessor

Successor: Find the minimal entry in the right sub-tree, *if* there is a right sub-tree. Otherwise find the first parent that the entry is in its left sub-tree.

Predecessor: Find the maximal entry in the left sub-tree, *if* there is a left sub-tree. Otherwise find the first parent that the entry is in its right sub-tree.

In either test, if the root node is reached, no predecessor/ successor exists.

Simple Insertion and Deletion

Insertion: Traverse the tree as you would when searching. When the required branch does not exist, attach the new entry at that location.

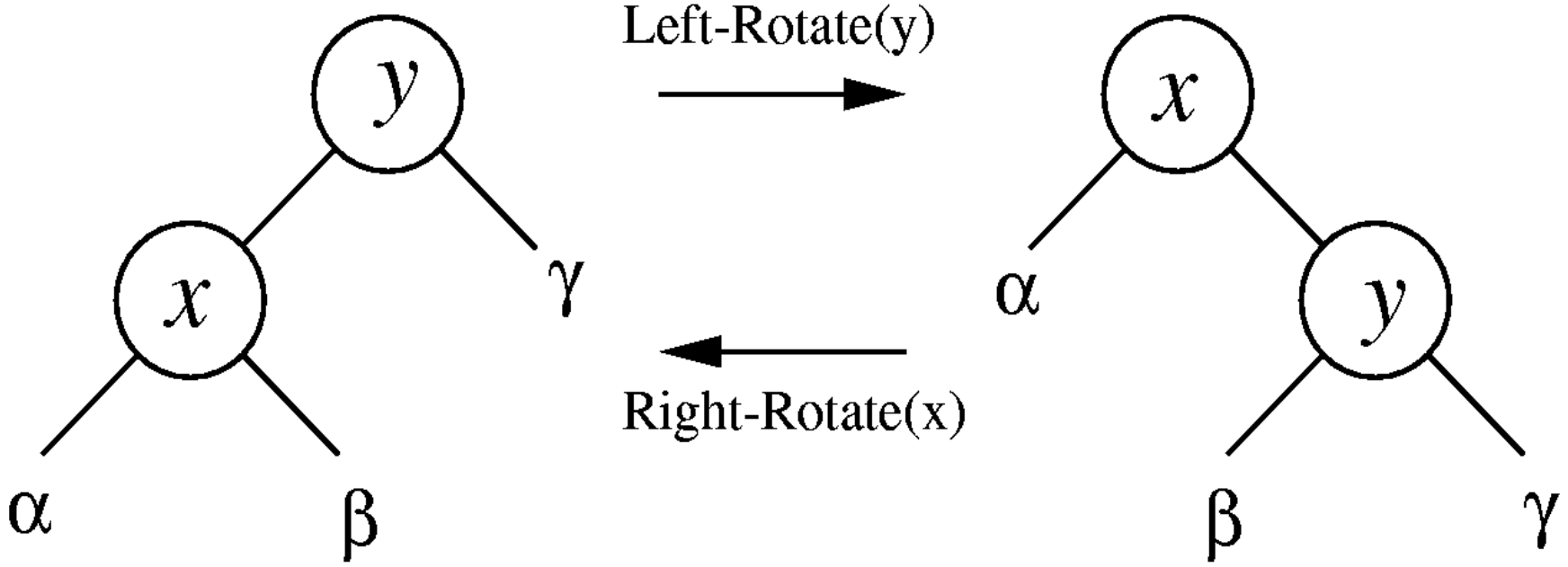
Deletion: Three possible cases exist:

a) *Entry is a leaf* : Just delete it.

b) *Entry has one child* : Remove entry replacing it with child.

c) *Entry had two children* : Replace entry with successor. Successor has at most one child; use step a or b on it.

Manipulating Search Trees



Tree-Balancing Algorithms

- AVL Trees
- Splay Trees
- B-Trees (e.g., 2-3 Trees and 2-3-4 Trees)

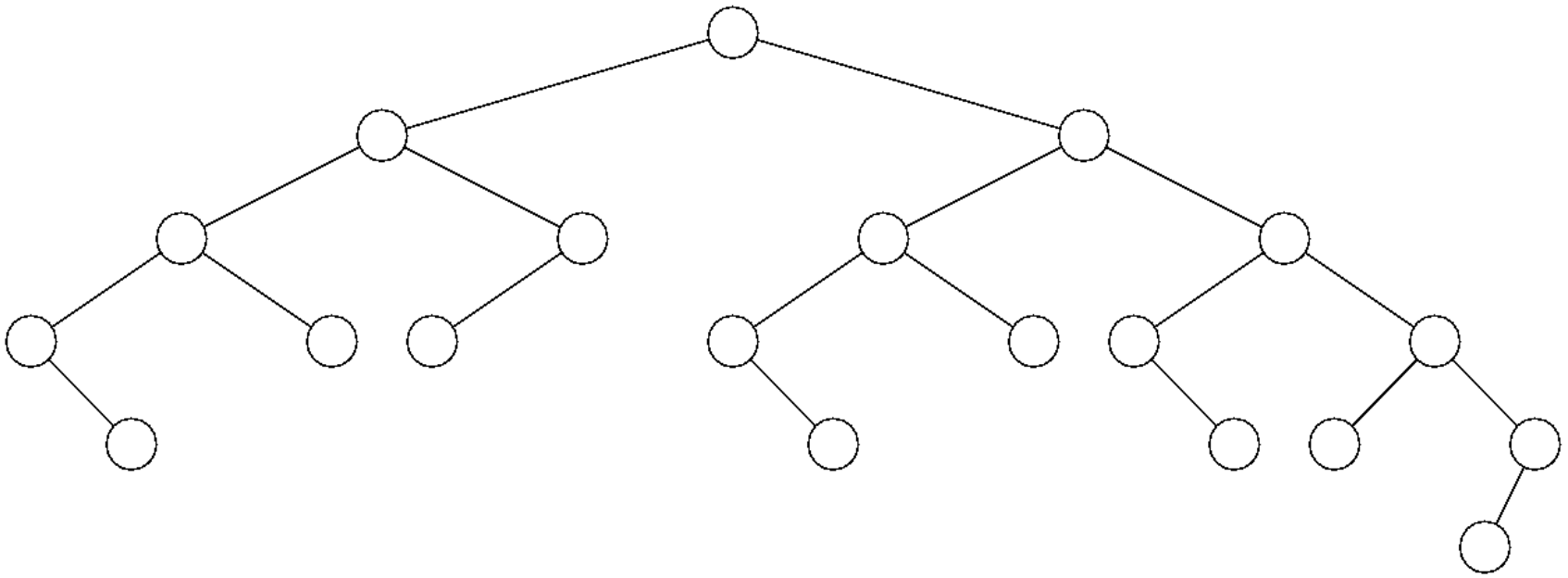
AVL Trees

The two sub-trees in an AVL tree differ in height by at most 1, and are in turn both AVL trees.

How can we be sure to maintain this property when inserting and deleting elements in the tree?

Does this guarantee us a “good” binary search tree?

AVL Trees

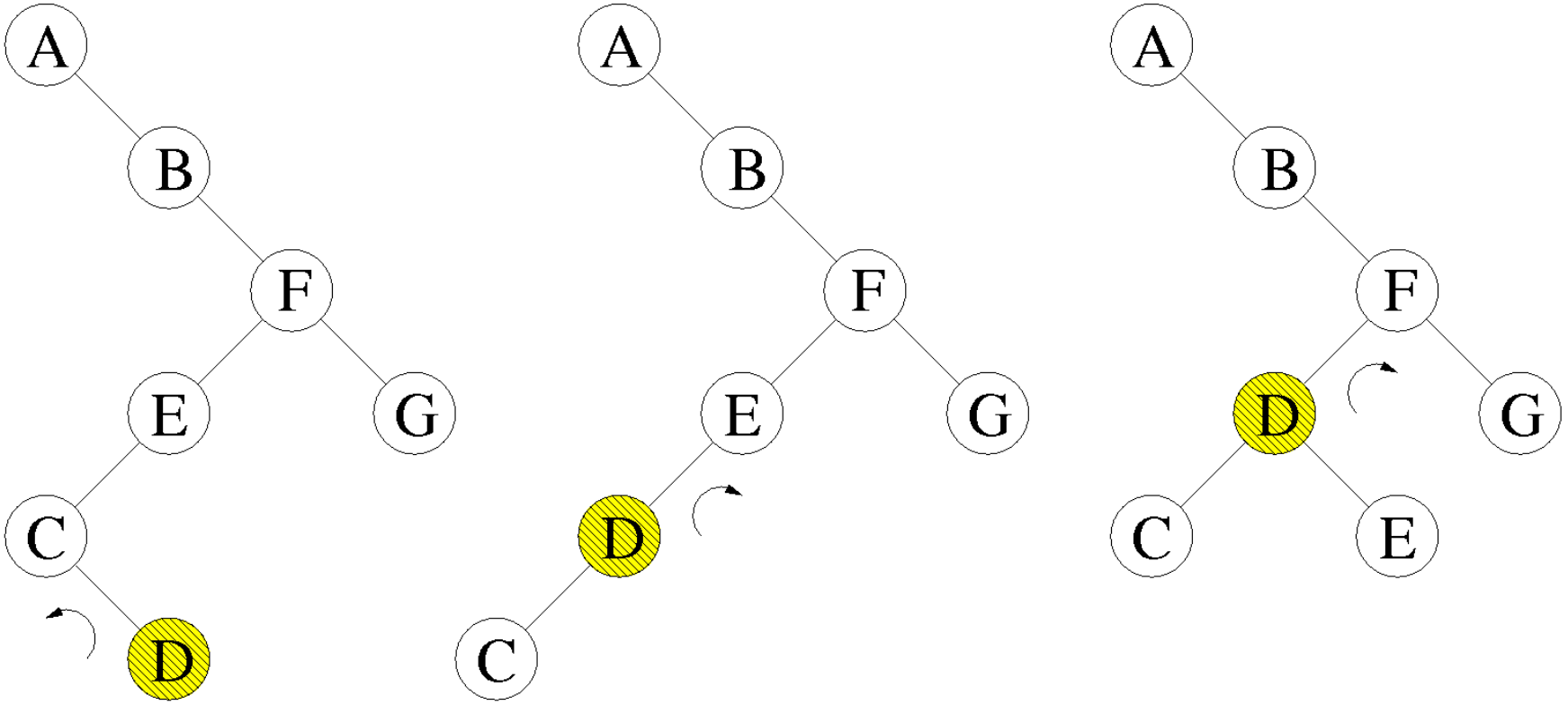


Splay Trees

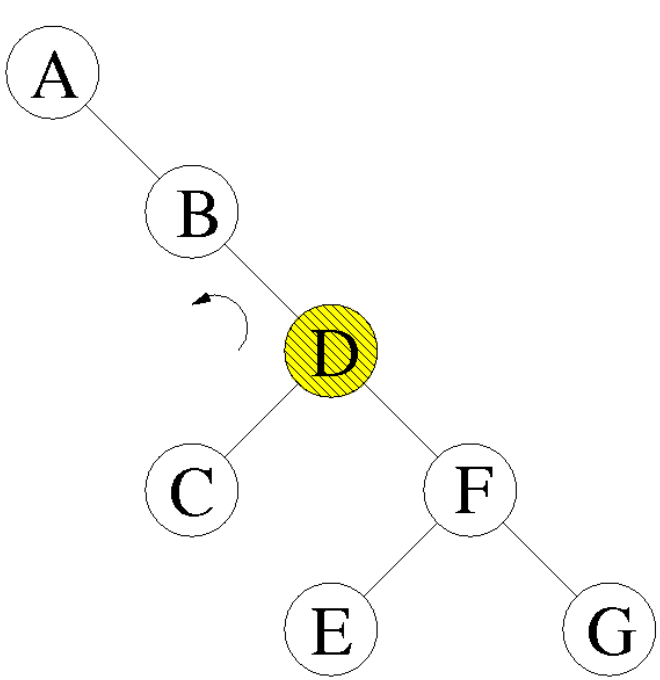
No adjustment is done in a splay tree when nodes are inserted or removed. All rotations occur within the *Search* function - the element being searched for is rotated to the root of the tree.

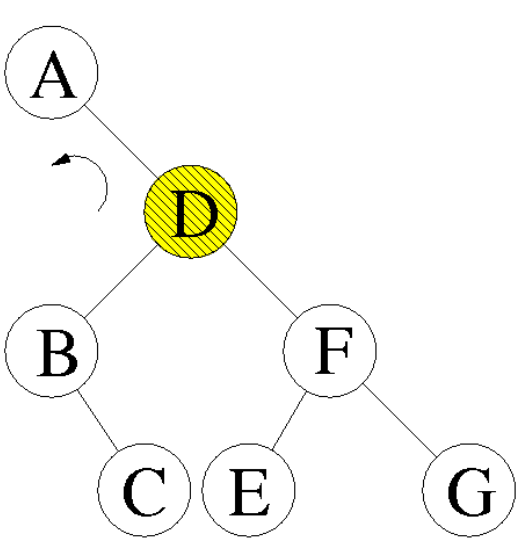
Initial searches in the tree may take O(n) time, but they will rapidly reduce to O(log n)

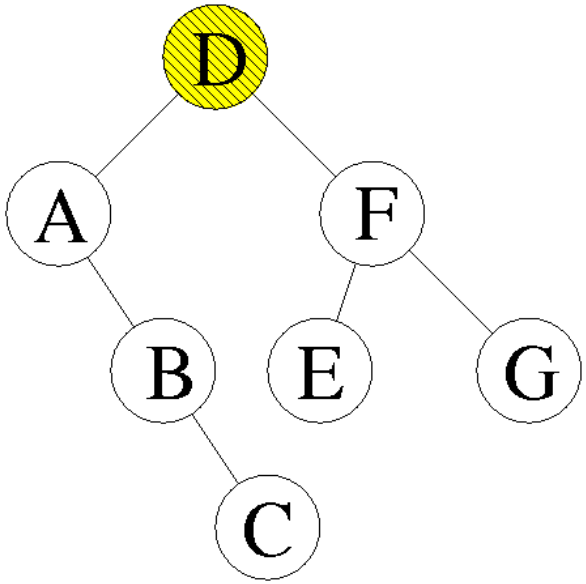
Splay Tree Example



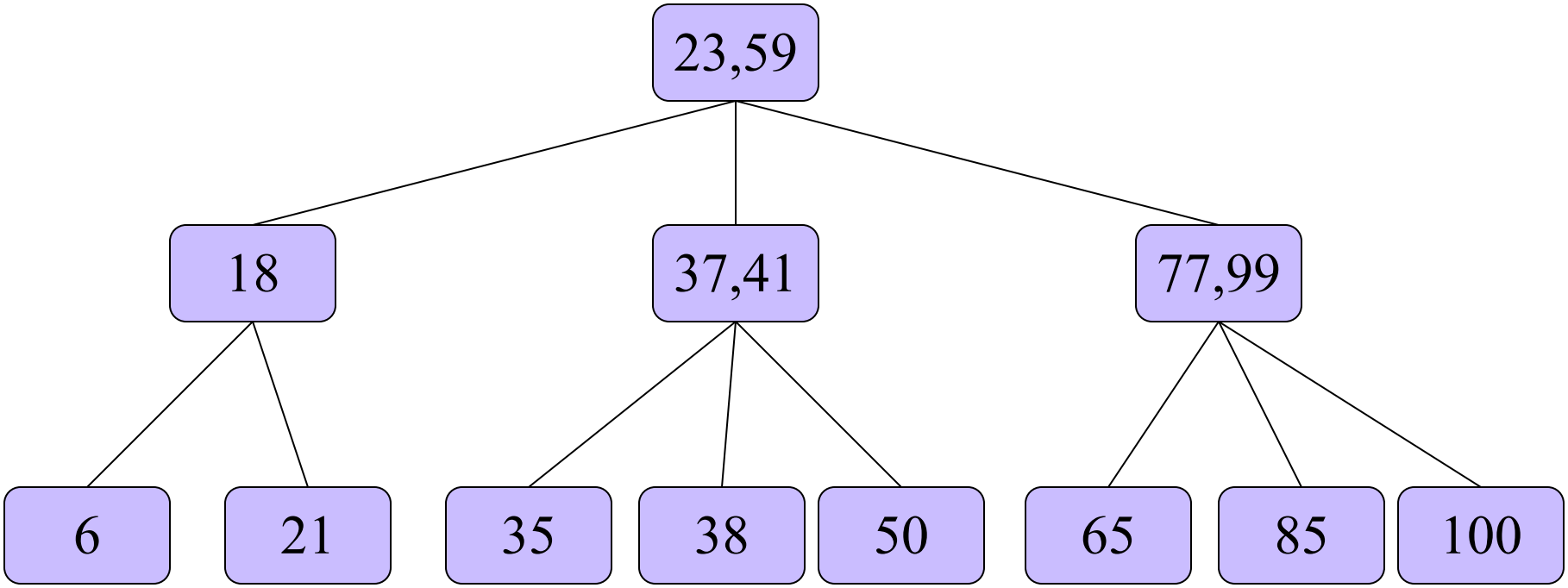
Splay Tree Example







B-Trees

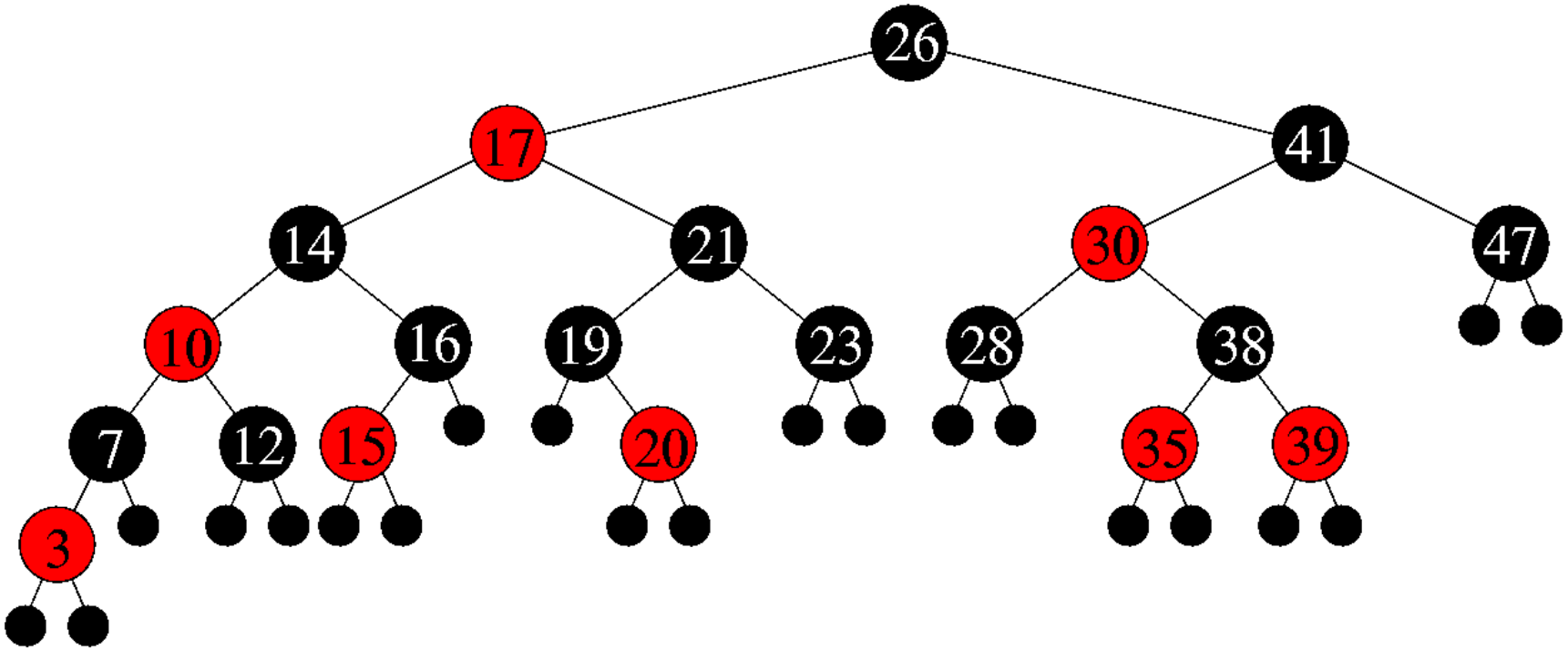


Red-Black Trees

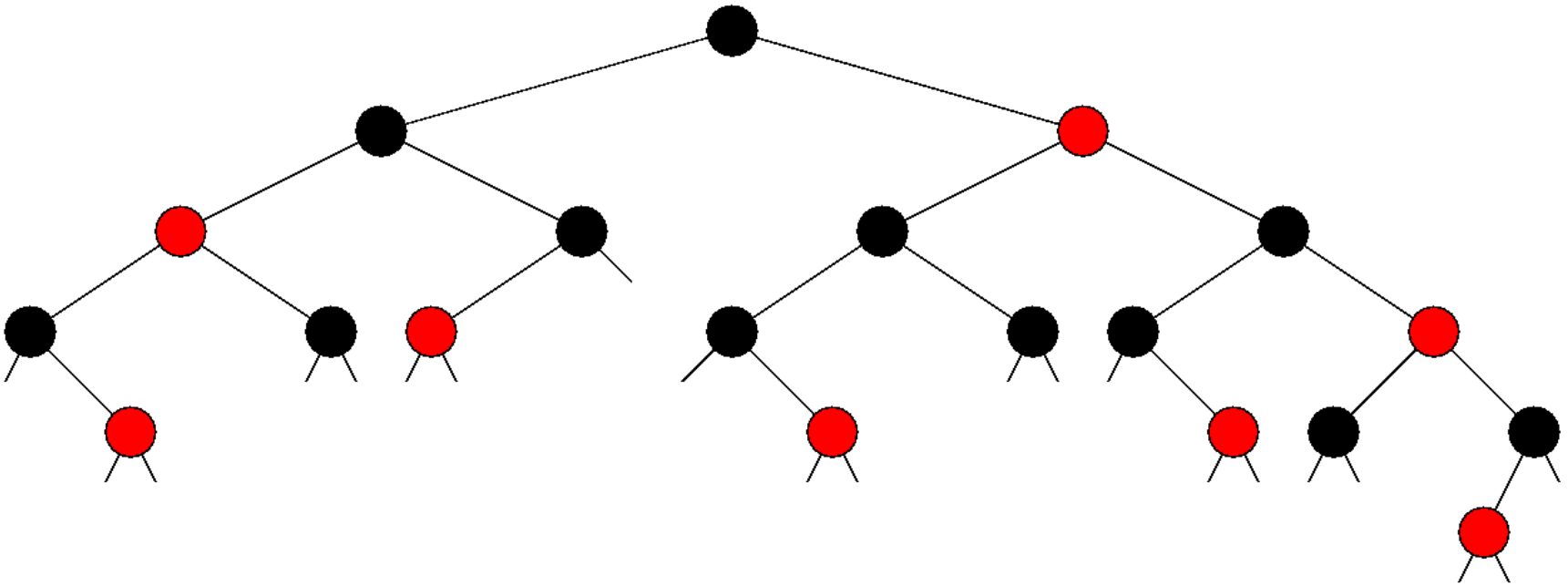
- All nodes in the tree are either **red** or **black**.
- Every leaf has NULL value and is colored **black**.
- The root node is colored **black**.
- All **red** nodes must have two **black** children.
- Every path from the root to a leaf must have the same number of **black** nodes.

How balanced of a tree will this produce? How hard will it be to maintain?

Example Red-Black Tree



AVL-Tree as Red-Black Tree



Data Structures: Worst Case Times

	Search	Insert	Delete	Index	Min	Max	Pred	Succ
Unsorted Array	O(n)	O(1)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)
Sorted Array	O(log n)	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)	O(1)
Unsorted List	O(n)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)	O(n)
Sorted List	O(n)	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)	O(1)
Binary Search Tree								
AVL Tree								
Hash Table								
Heap								

Data Structures: Worst Case Times

	Search	Insert	Delete	Index	Min	Max	Pred	Succ
Unsorted Array	O(n)	O(1)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)
Sorted Array	O(log n)	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)	O(1)
Unsorted List	O(n)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)	O(n)
Sorted List	O(n)	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)	O(1)
Binary Search Tree	O(n)	O(n)	O(n)	-	O(n)	O(n)	O(n)	O(n)
AVL Tree	O(log n)	O(log n)	O(log n)	-	O(log n)	O(log n)	O(log n)	O(log n)
Hash Table								
Heap								

Hash Tables

Hash tables are a *practical* way to maintain a dictionary.

Looking up a function in an array is a $\Theta(1)$ operation once you have the index *if you used a good hash function*. Hash functions map dictionary search terms to array indices.

What would a good hash functions look like? A bad hash function?

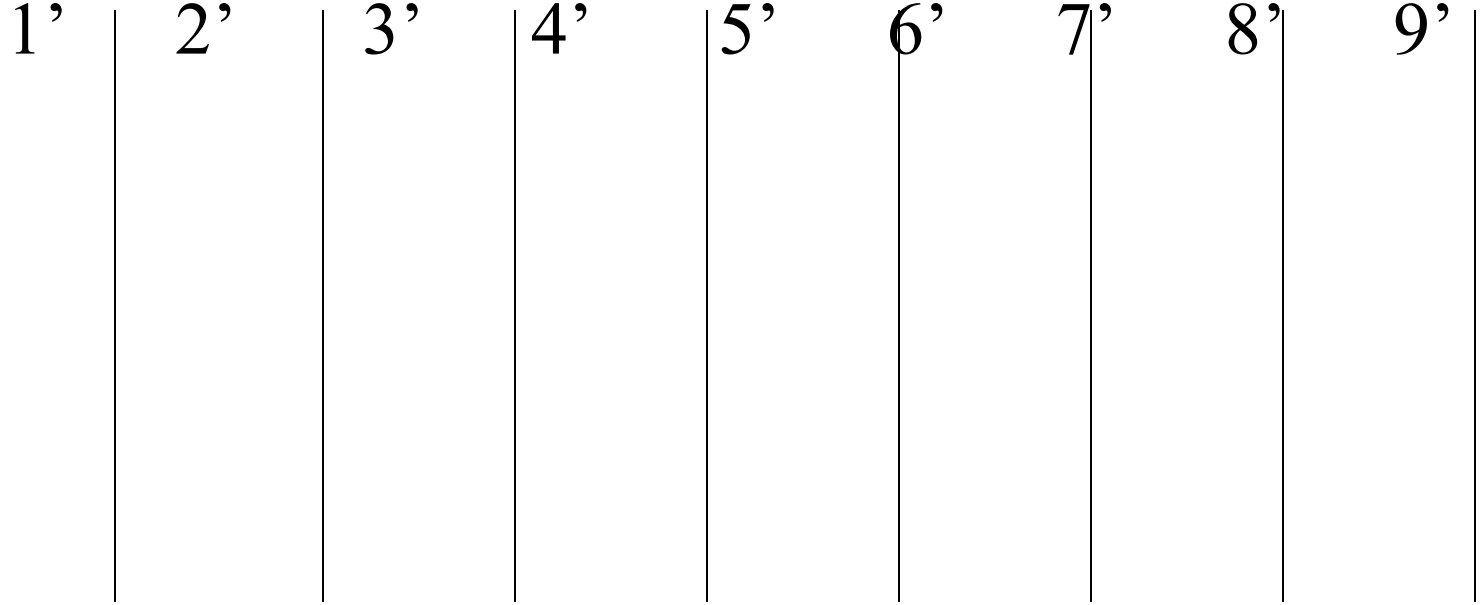
Hash Functions

A good hash function:

- Is cheap to evaluate.
- Tends to use all position 1..M with equal frequency.
- Puts similar keys into different parts of the table.

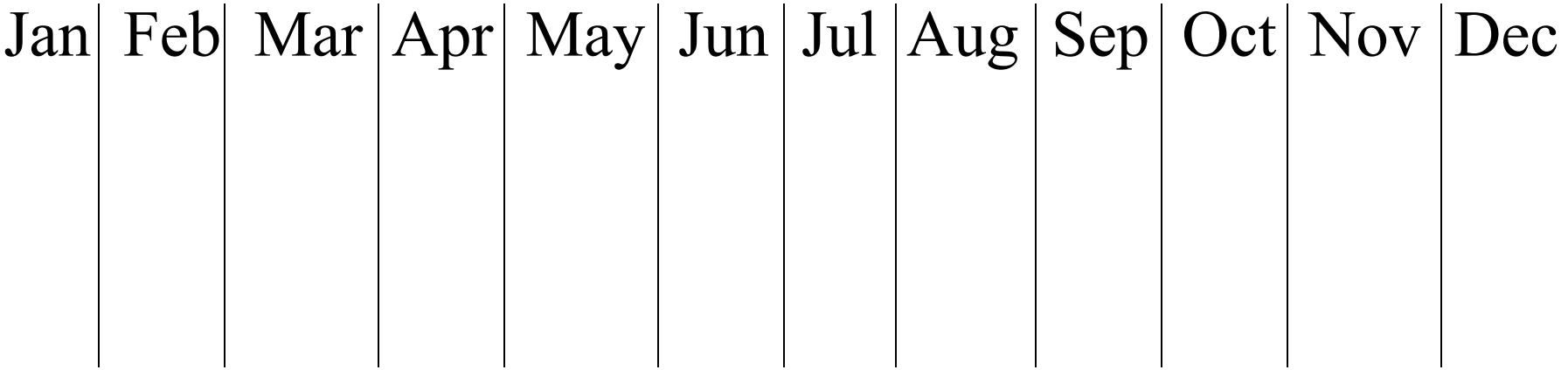
So, what functions should we avoid?

Hashing by Height



The Birthday Paradox

Even if we have a good function, we will still have collisions:



Data Structures: Worst Case Times

	Search	Insert	Delete	Index	Min	Max	Pred	Succ
Unsorted Array	O(n)	O(1)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)
Sorted Array	O(log n)	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)	O(1)
Unsorted List	O(n)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)	O(n)
Sorted List	O(n)	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)	O(1)
Binary Search Tree	O(n)	O(n)	O(n)	-	O(n)	O(n)	O(n)	O(n)
AVL Tree	O(log n)	O(log n)	O(log n)	-	O(log n)	O(log n)	O(log n)	O(log n)
Hash Table								
Heap								

Data Structures: Worst Case Times

	Search	Insert	Delete	Index	Min	Max	Pred	Succ
Unsorted Array	O(n)	O(1)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)
Sorted Array	O(log n)	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)	O(1)
Unsorted List	O(n)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)	O(n)
Sorted List	O(n)	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)	O(1)
Binary Search Tree	O(n)	O(n)	O(n)	-	O(n)	O(n)	O(n)	O(n)
AVL Tree	O(log n)	O(log n)	O(log n)	-	O(log n)	O(log n)	O(log n)	O(log n)
Hash Table	O(n)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Heap								

Data Structures: Worst Case Times

	Search	Insert	Delete	Index	Min	Max	Pred	Succ
Unsorted Array	O(n)	O(1)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)
Sorted Array	O(log n)	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)	O(1)
Unsorted List	O(n)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)	O(n)
Sorted List	O(n)	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)	O(1)
Binary Search Tree	O(n)	O(n)	O(n)	-	O(n)	O(n)	O(n)	O(n)
AVL Tree	O(log n)	O(log n)	O(log n)	-	O(log n)	O(log n)	O(log n)	O(log n)
Hash Table (good hash!)	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Heap								

Priority Queues

Insert(*x*) : Given an item *x*, insert it into the priority Queue.

Find-Maximum( ) : Return the item with the maximal priority.

Delete-Maximum( ) : Remove the item from the queue whose key is maximum.

Binary Heaps

A *binary heap* is defined to be a binary tree with a key in each node such that:

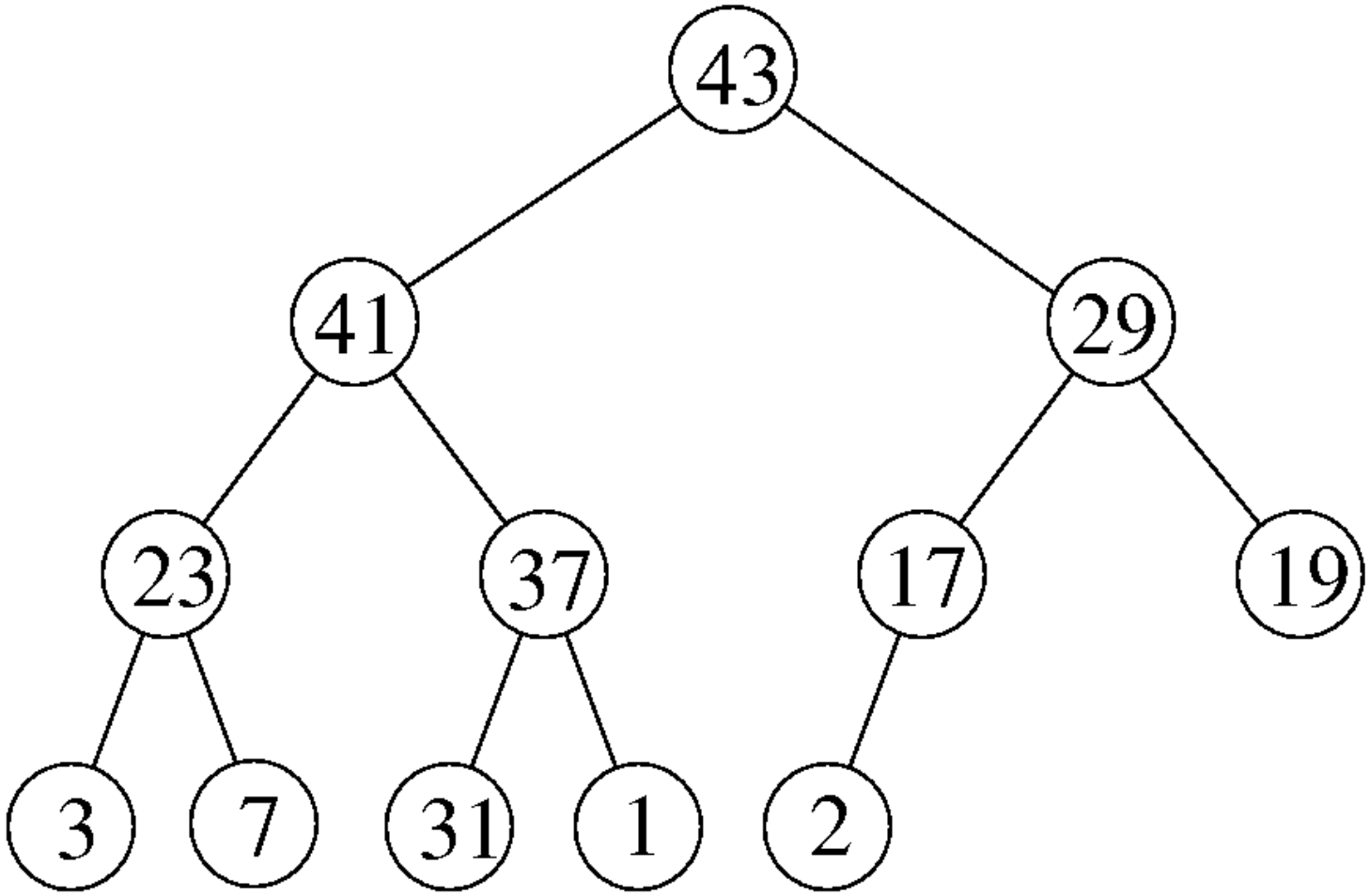
1: All leaves are on, at most, two adjacent levels.

2: All leaves on the lowest level occur to the left, and all levels except the lowest one are completely filled.

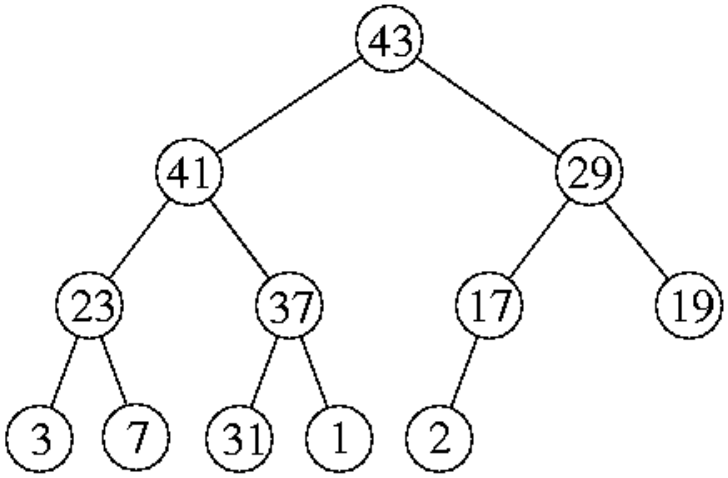
3: The key in the root is greater than all its children, and the left and right subtrees are again binary heaps.

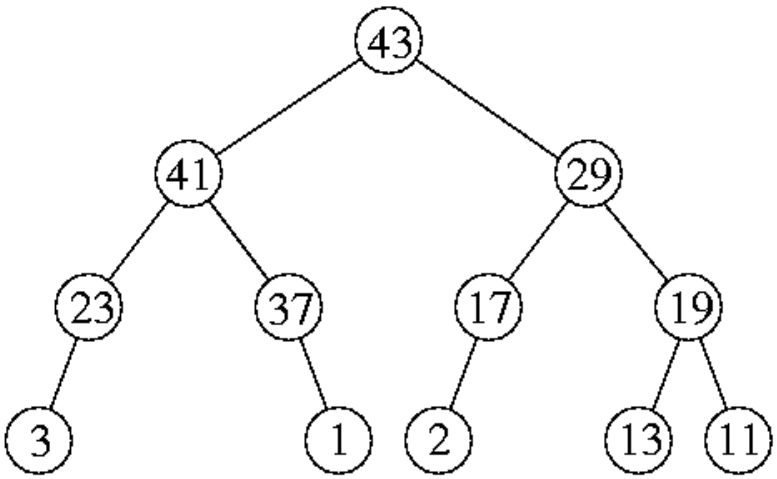
Conditions 1 and 2 specify shape of the tree, and condition 3 the labeling of the tree.

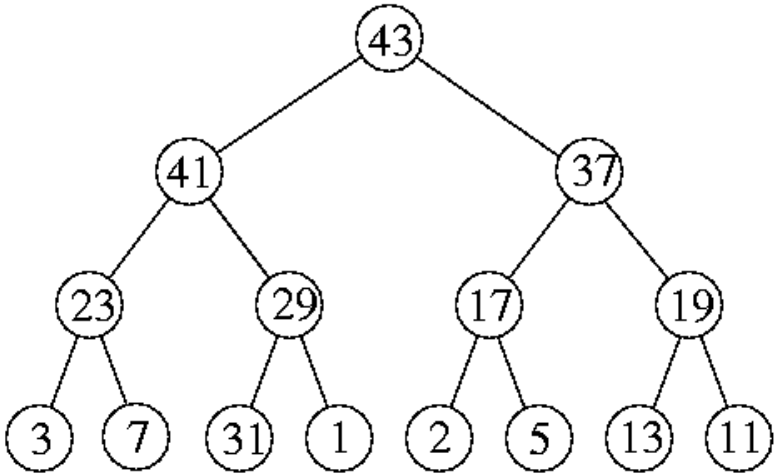
Example Heap

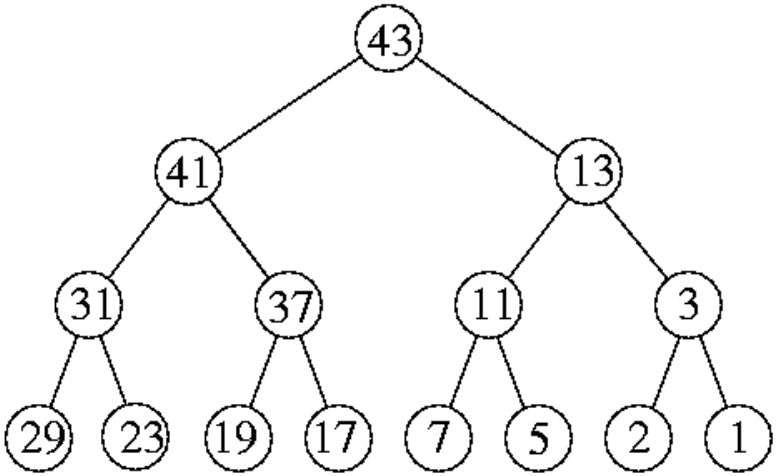


Are these legal?









Properties of Heaps

The ancestor relation in a heap defines a *partial order* on its elements, which means it is reflexive, anti-symmetric, and transitive.

Reflexive: x is an ancestor of itself.

Anti-symmetric: if x is an ancestor of y and y is an ancestor of x, then $x = y$.

Transitive: if x is an ancestor of y and y is an ancestor of z, x is an ancestor of z.

Partial orders can be used to model hierarchies with incomplete information or equal-valued elements.

Constructing Heaps

- Heaps can be constructed incrementally, by inserting new elements into the left-most open spot in the array.
- If the new element is greater than its parent, swap their positions and recur.
- The height h of an n element heap is bounded because:

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1 \geq n$$

so, $h = \lfloor \log n \rfloor$ and insertions take O(log n) time

Heapify

The bottom up insertion algorithm gives a good way to build a heap, but Robert Floyd found a better way, using a *merge* procedure called *heapify*.

Given two heaps and a fresh element, they can be merged into one by making the new entry the root and trickling down.

To convert an array of integers into a heap, place them all into a binary tree, and call heapify on each node.

How long would this take?

Heapify Example

Try to create a heap with the entries:

5, 3, 17, 10, 84, 19, 6, 22, 9

Heap Extract Max

if *heap-size*(A) < 1

then error “Heap Underflow”;

max = A[1];

A[1] = A[*heap-size*(A)];

heap-size(A)--;

Heapify(A, 1);

return *max;*

Data Structures: Worst Case Times

	Search	Insert	Delete	Index	Min	Max	Pred	Succ
Unsorted Array	O(n)	O(1)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)
Sorted Array	O(log n)	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)	O(1)
Unsorted List	O(n)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)	O(n)
Sorted List	O(n)	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)	O(1)
Binary Search Tree	O(n)	O(n)	O(n)	-	O(n)	O(n)	O(n)	O(n)
AVL Tree	O(log n)	O(log n)	O(log n)	-	O(log n)	O(log n)	O(log n)	O(log n)
Hash Table (good hash!)	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Heap								

Data Structures: Worst Case Times

	Search	Insert	Delete	Index	Min	Max	Pred	Succ
Unsorted Array	O(n)	O(1)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)
Sorted Array	O(log n)	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)	O(1)
Unsorted List	O(n)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)	O(n)
Sorted List	O(n)	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)	O(1)
Binary Search Tree	O(n)	O(n)	O(n)	-	O(n)	O(n)	O(n)	O(n)
AVL Tree	O(log n)	O(log n)	O(log n)	-	O(log n)	O(log n)	O(log n)	O(log n)
Hash Table (good hash!)	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Heap	O(n)	O(log n)	O(log n)	-	O(n)	O(1)	O(n)	O(n)

Data Structures: Worst Case Times

	Search	Insert	Delete	Index	Construct	Space	Annotations
Unsorted Array	O(n)	O(1)	O(1)	O(1)			
Sorted Array	O(log n)	O(n)	O(n)	O(1)			
Unsorted List	O(n)	O(1)	O(1)	O(n)			
Sorted List	O(n)	O(n)	O(1)	O(n)			
Binary Search Tree	O(n)	O(n)	O(n)	-			
AVL Tree	O(log n)	O(log n)	O(log n)	-			
Hash Table (good hash!)	O(1)	O(1)	O(1)	-			
Heap	O(n)	O(log n)	O(log n)	-			

Data Structures: Worst Case Times

	Search	Insert	Delete	Index	Construct	Space	Annotations
Unsorted Array	O(n)	O(1)	O(1)	O(1)	O(1)		
Sorted Array	O(log n)	O(n)	O(n)	O(1)	O(n log n)		
Unsorted List	O(n)	O(1)	O(1)	O(n)	O(n)		
Sorted List	O(n)	O(n)	O(1)	O(n)	O(n log n)		
Binary Search Tree	O(n)	O(n)	O(n)	-	O(n log n)		
AVL Tree	O(log n)	O(log n)	O(log n)	-	O(n log n)		
Hash Table (good hash!)	O(1)	O(1)	O(1)	-	O(n)		
Heap	O(n)	O(log n)	O(log n)	-	O(n)		

Data Structures: Worst Case Times

	Search	Insert	Delete	Index	Construct	Space	Annotations
Unsorted Array	O(n)	O(1)	O(1)	O(1)	O(1)	O(n)	
Sorted Array	O(log n)	O(n)	O(n)	O(1)	O(n log n)	O(n)	
Unsorted List	O(n)	O(1)	O(1)	O(n)	O(n)	O(n)	
Sorted List	O(n)	O(n)	O(1)	O(n)	O(n log n)	O(n)	
Binary Search Tree	O(n)	O(n)	O(n)	-	O(n log n)	O(n)	
AVL Tree	O(log n)	O(log n)	O(log n)	-	O(n log n)	O(n)	
Hash Table (good hash!)	O(1)	O(1)	O(1)	-	O(n)	O(n+m)	
Heap	O(n)	O(log n)	O(log n)	-	O(n)	O(n)	

Data Structures: Worst Case Times

	Search	Insert	Delete	Index	Construct	Space	Annotations
Unsorted Array	O(n)	O(1)	O(1)	O(1)	O(1)	O(n)	(none)
Sorted Array	O(log n)	O(n)	O(n)	O(1)	O(n log n)	O(n)	(none)
Unsorted List	O(n)	O(1)	O(1)	O(n)	O(n)	O(n)	list pointers
Sorted List	O(n)	O(n)	O(1)	O(n)	O(n log n)	O(n)	list pointers
Binary Search Tree	O(n)	O(n)	O(n)	-	O(n log n)	O(n)	left, right children
AVL Tree	O(log n)	O(log n)	O(log n)	-	O(n log n)	O(n)	children + subtree size
Hash Table (good hash!)	O(1)	O(1)	O(1)	-	O(n)	O(n+m)	full table + overflow lists
Heap	O(n)	O(log n)	O(log n)	-	O(n)	O(n)	(none)

Example Problems

a. You are given a pile of thousands of telephone bills and thousands of checks sent in to pay the bills. Find out who did not pay.

Example Problems

b. You are given a list containing the title, author, call number and publisher of all the books in a school library and another list of 30 publishing companies. Find out how many of the books in the library were published by each of those 30 publishers.

Example Problems

c. You are given all the book checkout cards used in the campus library during the past year, each of which contains the name of the person who took out the book. Determine how many distinct people checked out at least one book.

Specialized Data Types

- Strings
- Geometric shapes
- Graphs
- Sets
- Schedules

Next time:
Workshop on Customized Data Structures