

Data Structures and Algorithms

*Presented by:
Bloomberg Finance L.P.
R&D Training*

Learning Objectives

◆ Data Structures

- *Understand how a problem's characteristics affect the way data must be organized for an efficient solution.*
- *Understand the features and drawbacks of commonly used data structures*
- *Be able to choose the right data structures based on a problem's characteristics*

◆ Algorithms

- *Learn how to measure and compare the performance and efficiency of algorithms*
- *Understand "divide and conquer" as a design strategy, and how to apply it through recursion*
- *Be able to choose search and sorting algorithms appropriate to a problem's domain.*

Course Outline

◆ Preliminaries

- > *Motivation*
- > *Criteria for comparing data structures and algorithms*

◆ Data Structures

- > *Arrays and Linked Lists*
- > *Graphs and Trees*
- > *Abstract Data Types: Stacks, Queues, and Maps*
- > *Binary Trees and Hash Tables*
- > *Survey of Balanced Trees*

◆ Algorithms

- > *Divide and Conquer Algorithms*
- > *Searching*
- > *Sorting*

Preliminaries



- ◆ **Motivation**
- ◆ **Criteria for comparing data structures and algorithms**

Example

Problem:

You are given a dictionary with 70,000 words.

Write a program that takes a string and generates all of its anagrams contained in the dictionary.

- ♦ **input:** eat
output: ate, eat, tea
- ♦ **input:** read
output: dare, dear, read

Impatient Hacker Solution

```
-----  
input x  
for each possible rearrangement r of x  
    look up r in dictionary  
    if found, output r  
-----
```

- ◆ Assume that one basic operation takes roughly 4×10^{-10} seconds (≈ 2.4 GHz).
- ◆ Using binary search, looking up one word takes roughly 0.025 microseconds ($= 2.5 \times 10^{-8}$ sec).
- ◆ How long will this algorithm take to determine all anagrams of the word **deinstitutionalization** ?

Impatient Hacker Solution (cont.)

- ♦ The word `deinstitutionalization` has 22 letters (some repeated).

- ♦ There are roughly

$$\frac{22!}{5! \times 4! \times 3! \times 2! \times 2!} \approx 1.63 \times 10^{16}$$

different ways to arrange the letters.

- ♦ So the total running time will be

$$1.63 \times 10^{16} * 2.5 \times 10^{-8} \text{ sec} = 407\,500\,000 \text{ sec}$$

seconds. That's almost **thirteen years!**

Expert Hacker Solution

```
-----  
input x  
x' := sorted letters of x  
for each word y in dictionary  
    check if x and y are anagrams  
        by sorting letters in y  
        and comparing to x'  
    if yes, output y  
-----
```

- ◆ Each iteration takes roughly **0.004 microseconds**.
- ◆ We need to run **70000 iterations**, so the total execution time is about **0.28 milliseconds**.

Can we do better?

- ◆ The expert hacker is content with this solution and takes a vacation in the Bahamas.
- ◆ It's good, but can we do better? How much better?
- ◆ How about **0.025 microseconds**?
 - *About 11 000 times faster than the expert hacker solution!*

Master Solution

Idea:

Don't keep the dictionary in alphabetical order, but sort it by the words' signatures. The signature of a word is the alphabetically sorted list of all the letters it contains.

Word	Signature
dare	ader
dear	ader
dog	dgo
lure	elru
god	dgo
read	ader
rule	elru



Word	Signature
dare	ader
dear	ader
read	ader
dog	dgo
god	dgo
lure	elru
rule	elru

Master Solution (Cont.)

input **x**

use binary search to find signature
of **x** in dictionary

output the entire anagram group

- input: rule
- signature: elru
- output: lure, rule

Word	Signature
dare	ader
dear	ader
read	ader
dog	dgo
god	dgo
lure	elru
rule	elru

Master Solution (Cont.)

input **x**
 use binary search to find signature
 of **x** in dictionary
 output the entire anagram group

- ◆ Sorting the dictionary once by signatures takes less than **1 second**.
- ◆ The computation for one input is essentially just a binary search which takes **0.025** microseconds.

Preliminaries

- ◆ Motivation



- ◆ Criteria for comparing data structures and algorithms

Selecting a Data Structure or Algorithm

◆ Performance:

- *Expected Runtime (= Time Complexity)*
- *Memory Requirements (=Space Complexity)*

◆ Scalability

◆ Ease of implementation

- *Availability*
- *Maintainability*
- *Familiarity*

◆ Consistency with existing code

◆ Monetary cost



Sometimes the criteria are contradictory.
Look at the requirements!

Performance Measurements

- ◆ Performance is measured relative to the size of the input
 - “Size of the input” usually refers to the number of pieces of data.
 - e.g., “How many comparisons are needed to sort n numbers”
- ◆ Performance measurements (usually) provide a rough approximation rather than an exact number
 - e.g., “Order of $n \cdot \log(n)$ comparisons required to sort n numbers”
- ◆ **Big O-Notation** provides a formal framework for comparing measurements

$O(1) < O(\log n) < O(n) < O(n \cdot \log n) < O(n^2) < O(n^3) < \dots < O(2^n) < \dots$

Constant

Logarithmic

Linear

Quadratic

Exponential

Time Complexity

- ◆ Measures the number of operations
- ◆ The type of operation depends on the problem domain.
 - *Comparisons (for searching)*
 - *Moving / Copying a single data item (for sorting)*
 - *Arithmetic (for numeric problems)*
- ◆ The time complexity of an **algorithm** measures how long that algorithm takes
 - *“The selection sort algorithm requires $O(n^2)$ comparisons to sort n values”*
- ◆ If a **problem** has a certain complexity, it cannot be solved any faster
 - *“It is impossible to sort n numbers with less than $O(n \cdot \log n)$ comparisons”*

Space Complexity

- ◆ Measures the number of data items that need to be stored.
- ◆ Approximates how much memory is required by an algorithm or data structure.

Average Case vs. Worst Case Complexity

- ◆ **Average-Case Complexity** provides a performance estimate sufficient for most circumstances
 - > *“Quicksort usually requires $O(n \cdot \log n)$ comparisons”*
 - > *Insufficient for critical algorithms in real-time environments (... “emergency shut-off in nuclear power plant” ...)*
- ◆ **Worst-Case Complexity** provides a minimum performance guarantee that will be met for all circumstances
 - > *“In the worst case, Quicksort requires $O(n^2)$ comparisons”*
- ◆ Often, worst case and average case performance are identical
- ◆ Worst-Case Analysis of algorithms / data structures is usually easier than Average-Case Analysis

Data Structures



- ◆ **Arrays and Linked Lists**
- ◆ **Graphs and Trees**
- ◆ **Binary Trees and Hash Tables**
- ◆ **Abstract Data Types: Stacks, Queues, and Maps**
- ◆ **Survey of Balanced Trees**

Arrays

- ◆ Array elements are

- *Fixed in number (usually)*
- *Stored contiguously*
- *Accessed via an **index***

12	4	19	34	1	4			
----	---	----	----	---	---	--	--	--

- ◆ Arrays provide **random access** to their elements

- *Time required to access any element is the same*

- ◆ Arrays provide **sequential access** to their elements

- *Can start at an element and step through adjacent elements in sequence*

Multi-dimensional and Dynamic Arrays

◆ Multi-dimensional arrays

➤ *Example: A table is a two-dimensional array.*

- It has two indices, one for the row and one for the column.

◆ Some programming languages allow arrays to grow dynamically at runtime.

➤ *You can allocate a minimal amount of memory at startup*

➤ *You can allocate more elements when the amount of data to be stored increases*

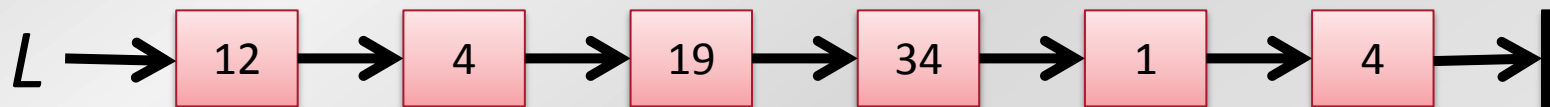
➤ *Dynamically changing the size of the array can be **expensive**.*

- When it grows too large, the contents must be copied to a different memory location.

Linked Lists

◆ A **Linked List** stores its content as a sequence of nodes

- *Each element is stored in its own memory location (a **node**)*
- *Each node also stores the location of the next node in the list.*



◆ **Benefits:**

- *Can easily grow and shrink during its lifetime*
 - Reduces wasted memory
 - Reduces unnecessary complexity
- *Elements can be re-arranged easily by “re-pointing” the locations.*

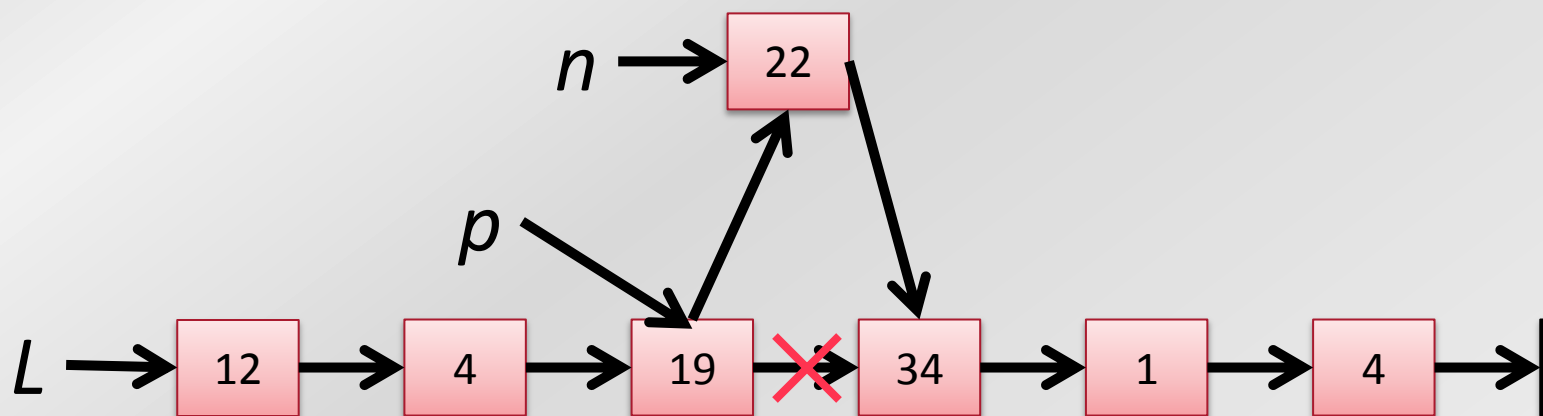
◆ **Drawbacks:**

- *Sequential access to elements – on average this requires $O(n)$ operations. No random access.*

Insertion Into a Linked List

◆ Insertion at an arbitrary position in a linked list is efficient

- *No data must be moved*
- *Only the pointers to locations must be adjusted*



Arrays and Linked Lists Comparisons

- ◆ **Sequential organization of an array's elements is implicit (the index corresponds with the sequence position)**
- ◆ **In a linked list, sequential organization is explicit - each element contains a link to the next node. The actual location is irrelevant.**
- ◆ **Inserting and deleting elements in a linked list is more efficient than in an array. Changing links is computationally cheap, but shifting array elements isn't.**
- ◆ **Searching for a value in an array can be more efficient than in a linked list. We'll see why.**

Data Structures

- ◆ Arrays and Linked Lists



- ◆ Graphs and Trees

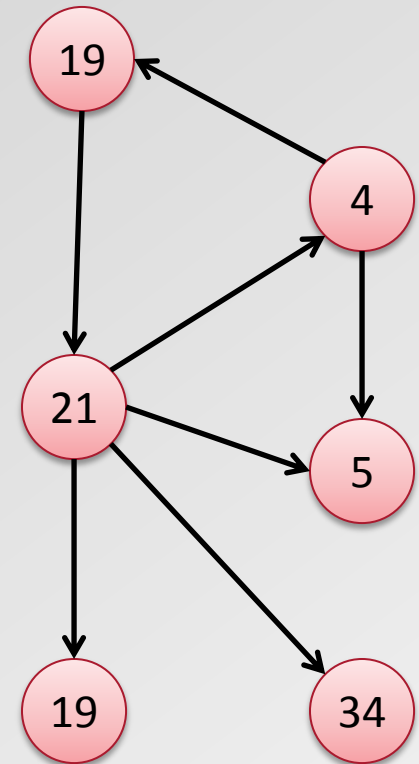
- ◆ Binary Trees and Hash Tables

- ◆ Abstract Data Types: Stacks, Queues, and Maps

- ◆ Survey of Balanced Trees

Graphs

- ◆ A **graph** contains
 - A set of **nodes**, and
 - A set of **edges** connecting the nodes.
- ◆ Graphs are used to model complex problems:
 - *Example: Shipping Routes, Network Topologies, Behavioral Associations, etc.*
- ◆ A graph is **directed** if an edge is directional
- ◆ A graph is **undirected** if no edges are directional
- ◆ A **cycle** is a sequence of adjacent edges with the same start and end point.
- ◆ A **Directed Acyclic Graph (DAG)** is a directed graph without cycles



Tree

♦ A **tree** is a directed acyclic graph where

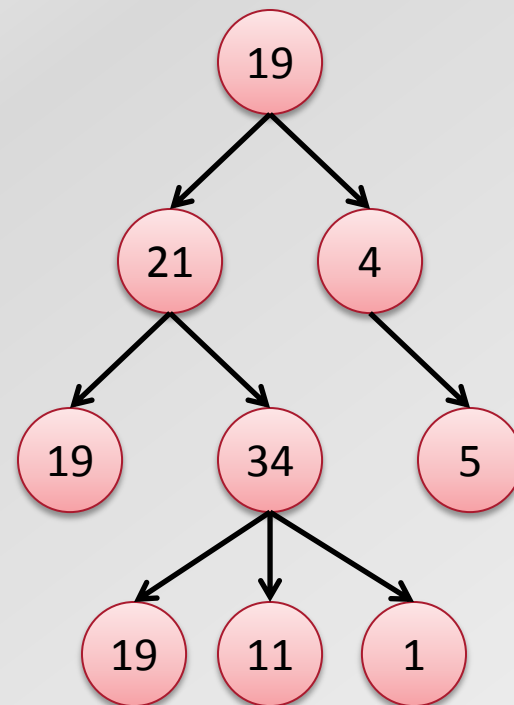
➤ *Exactly one node has no in-pointing edges (the **root**)*

➤ *All other nodes have exactly one in-pointing edge*

♦ So, for every node **v** there is a unique path from the root to **v**.

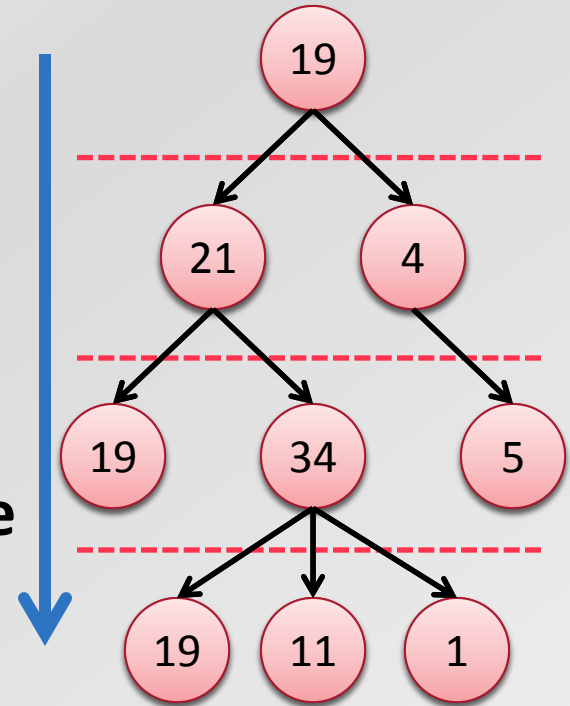
♦ Trees are used to model hierarchical relations:

➤ *Family trees, organizational charts, the grammatical structure of a sentence, etc.*



Tree Terminology

- ◆ **Leaf**: node with no outgoing edges.
- ◆ **Child**: node to which the outgoing edges point
- ◆ **Parent**: the node from which an incoming edge originates.
- ◆ **Subtree of a node**: the nodes that can be reached from that node.
- ◆ **Levels** of a tree: the nodes at the same distance from the root
- ◆ The **depth / height** of a tree is the length of the longest path from the root to any leaf.



Trees *(continued)*

- ◆ **Branching factor**

Number of children a node may have, which dictates how fast the tree will branch out as nodes are inserted.

- ◆ **Binary** trees have a branching factor of 2.

Data Structures

- ◆ Arrays and Linked Lists

- ◆ Graphs and Trees



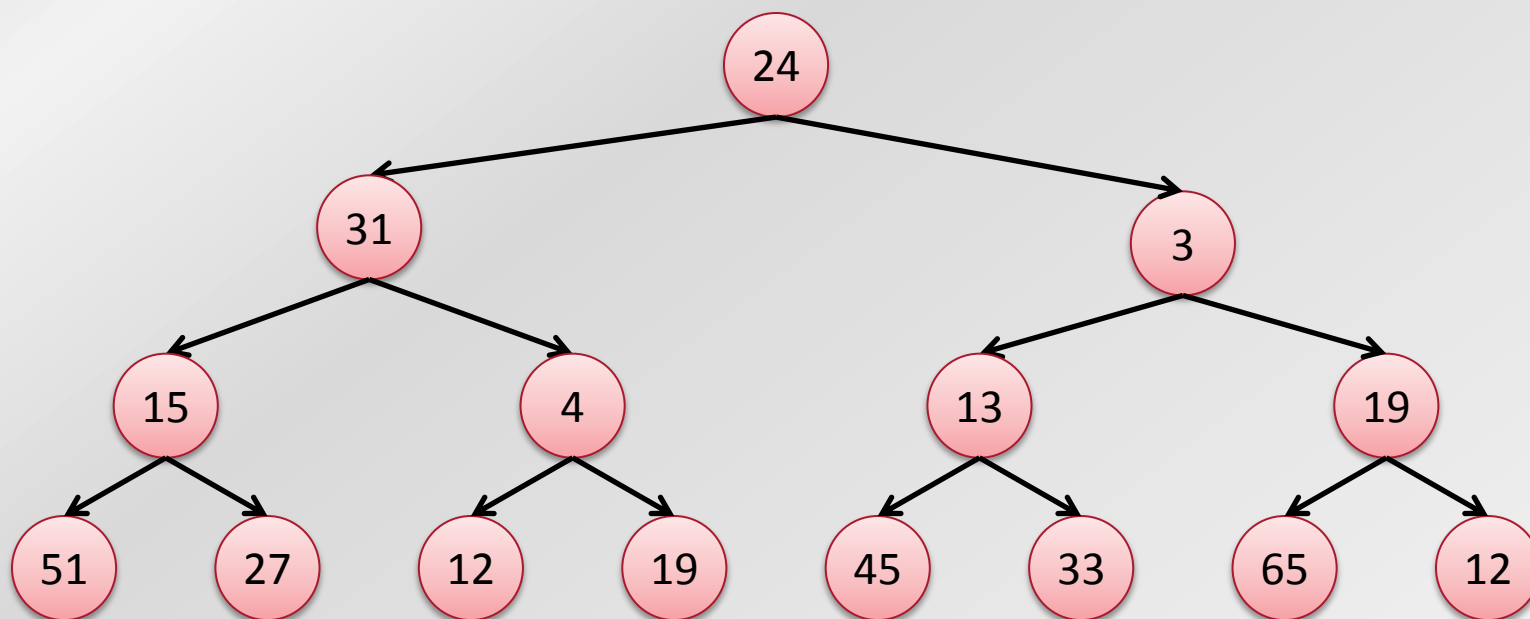
- ◆ Binary Trees and Hash Tables

- ◆ Abstract Data Types: Stacks, Queues, and Maps

- ◆ Survey of Balanced Trees

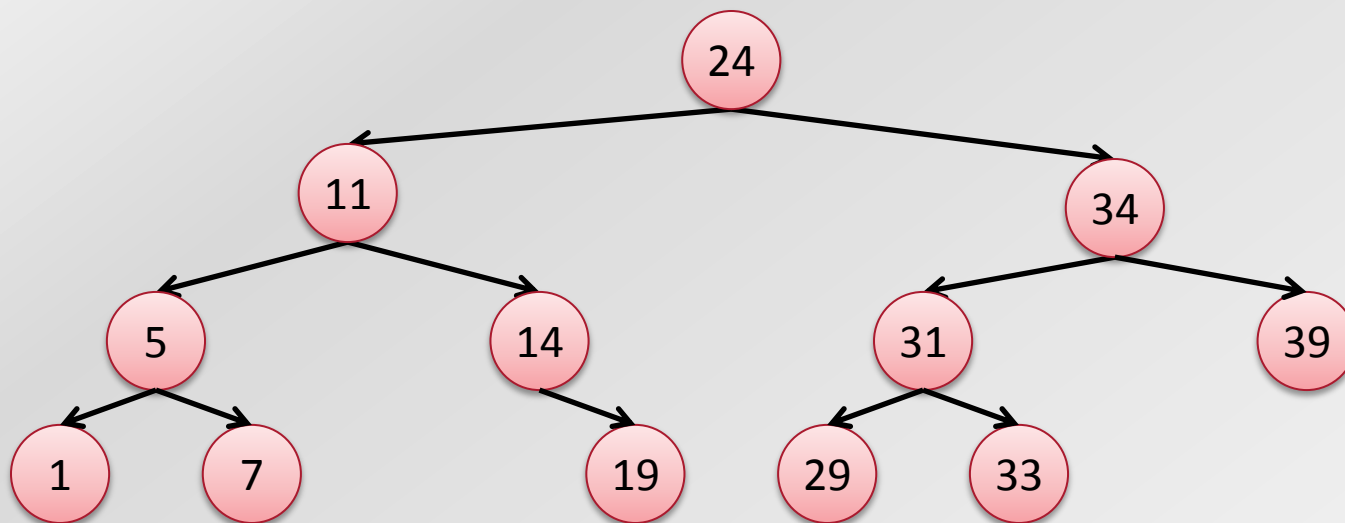
Binary Trees

- ◆ Every node has exactly two or zero children.
- ◆ A binary tree is **complete** if every level has as many nodes as possible



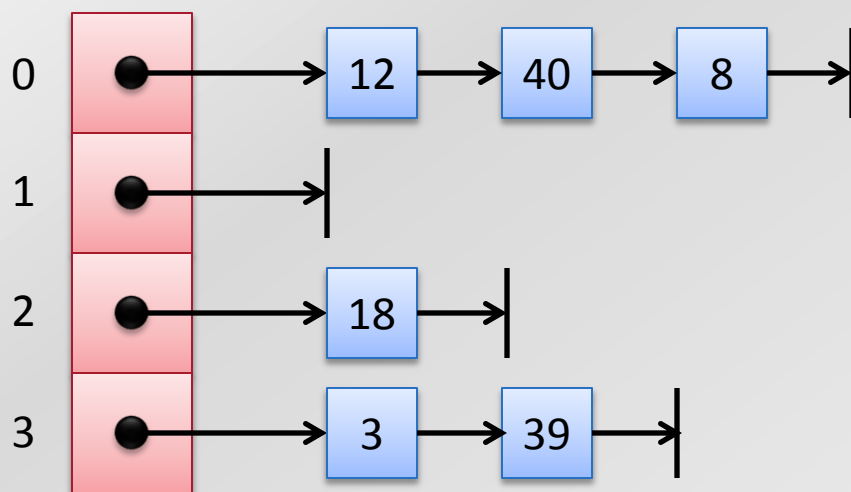
Binary Search Trees

- ◆ Every node **v** contains data and satisfies the following:
 - **v** has *no more than two children*
 - The data in every node in **v**'s left subtree is \leq v's data
 - The data in every node in **v**'s right subtree is $>$ v's data
- ◆ Binary Search Trees are typically *balanced*: the left and right subtrees have (roughly) the same number of nodes



Hash Tables

- ◆ A **hash table** stores data in an array of (short) linked lists called **buckets**.
- ◆ The hash table distributes the values into the buckets based on a **hash value** computed by a **hash function**.
- ◆ A good hash function ensures an even distribution of the values into the buckets



Simple example of a hash table.

It hashes the values based on their modulo 4 remainder

Hash Table vs. Binary Search Tree

Performance	Hash Table	Binary Search Tree
Insertion (worst)	$O(N)$	$O(N)$
Insertion	$O(1)$	$O(\log N)$
Lookup (best)*	$O(1)$	$O(\log N)$
Ordered Traversal	Not defined. Retrieval by key only (but depends on implementation)	"in-order" possible
Extra work for duplicate keys?	Yes	Yes
Data worst case	All data hashes to same value.	Pre-sorted. Duplicate keys.
Dynamic Memory Allocation?	Yes, but complicates implementation.	Nodes usually allocated dynamically (to simplify implementation).

Data Structures

- ◆ Arrays and Linked Lists

- ◆ Graphs and Trees

- ◆ Binary Trees and Hash Tables



- ◆ Abstract Data Types: Stacks, Queues, and Maps

- ◆ Survey of Balanced Trees

Abstract Data Types

- ◆ An **Abstract Data Type (ADT)** is characterized by
 - > *Data*
 - > *A collection of operations on the data (the interface)*
- ◆ Any specific implementation of an ADT must ensure the interface provides the expected functionality.
- ◆ Users of the ADT should not rely on how the interface is actually implemented, since the implementation could change. (The interface itself must *not* change)
- ◆ We will discuss the following Abstract Data Types
 - > *Stacks*
 - > *Queues*
 - > *Maps*

PushDown Stacks

◆ A **stack** is an Abstract Data Type supporting two operations:

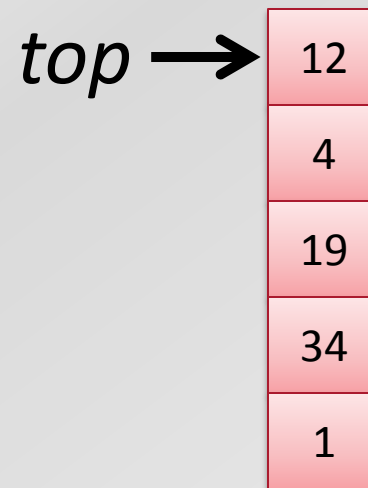
- **Push** an item on the top of the stack
- **Pop** an item off the top of the stack

◆ **LIFO** ("**L**ast **I**n **F**irst **O**ut")

- *Pop will always return the last item pushed*

◆ **Examples:**

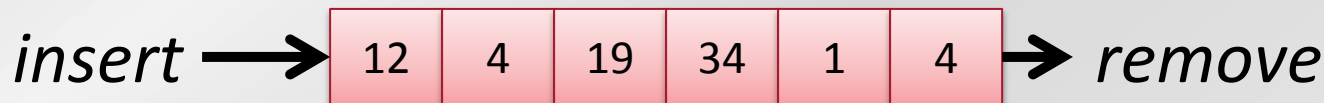
- *Evaluation of arithmetic expressions*
- *Parsing of text*
- *Function calls in many programming languages*



Queues

♦ A **queue** is an ADT providing two operations:

- *Insert an item at the beginning.*
- *Remove an item from the end.*



♦ **FIFO** ("**F**irst **I**n **F**irst **O**ut")

- *"First come first serve"*

♦ **Examples:**

- *Real-time Data Processing*
- *Network Routing*
- *Process Scheduling*

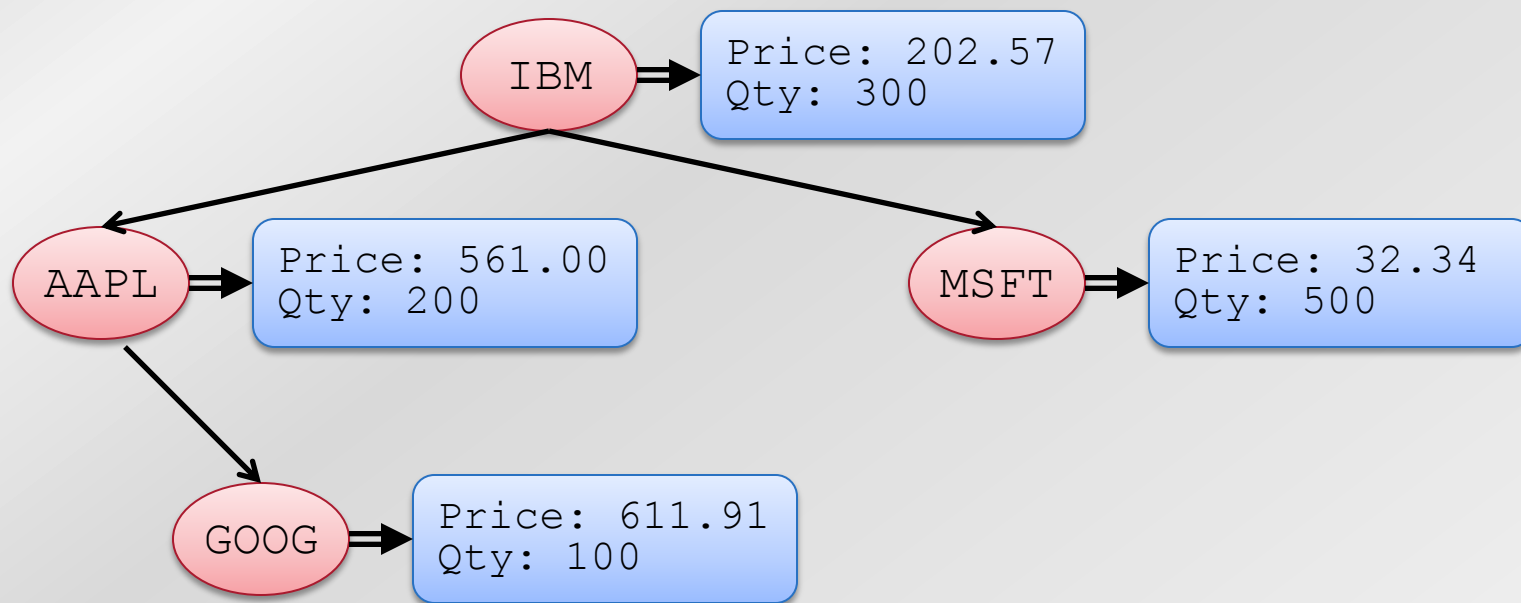
Maps

- ◆ A **map** is an ADT that stores data as an association between a **key** and a **value**.
 - *Maps are also called **Associative Arrays** or **Dictionaries**.*
 - *Provided in many modern programming languages (e.g. Java, C++ STL, Python, Perl, ...).*
- ◆ **Maps support the following operations:**
 - *Data access based on the key, for reading and writing*
 - *Inserting and Deleting entries based on their key.*
- ◆ **Maps can internally be implemented as**
 - *Binary Search Trees*
 - *Hash Tables*

Maps - Example


- ◆ **Example:** A map, internally implemented as a binary search tree, storing portfolio positions by mapping

Ticker to PortfolioData



- ◆ Data about a portfolio can be quickly retrieved based on the ticker.

Data Structures

- ◆ **Arrays and Linked Lists**
- ◆ **Graphs and Trees**
- ◆ **Binary Trees and Hash Tables**
- ◆ **Abstract Data Types: Stacks, Queues, and Maps**
-  ◆ **Survey of Balanced Trees**

An improvement for search trees?

- ◆ Binary search trees achieve their optimal performance, when the depth is small compared to the number of nodes.
- ◆ In the real world this is not automatically guaranteed:
 - *The ordering upon insertion might yield a deep but sparse tree*
 - E.g., inserting a sorted list of values yields a linear list
 - *Repeated additions and deletions of data can also change the shape of the tree.*
- ◆ Binary search trees are good, but how can we increase the likelihood of having the “best case” data structure?

Balanced Trees

- ◆ **Balanced Tree:** a tree where no leaf is much farther away from the root than any other leaf.
 - *Minimize the height of the tree*
 - By re-organizing after each insertion or deletion (**balancing**)
 - *Minimize search time*
- ◆ Different balancing schemes allow different definitions of "much farther" and different amounts of work to keep them balanced.
- ◆ The tree is balanced through **rotations**, after insertions or deletions.

Tree Rotation

- ◆ Switches children and parents among two or three adjacent nodes to restore balance to a tree.
- ◆ Useful to maintain a short or ``well-balanced'' tree in which searching for a key takes little time.
- ◆ **Left-rotation:** moving nodes to the left.
- ◆ **Right-rotation:** moving nodes to the right.
- ◆ See an animation [here](#).

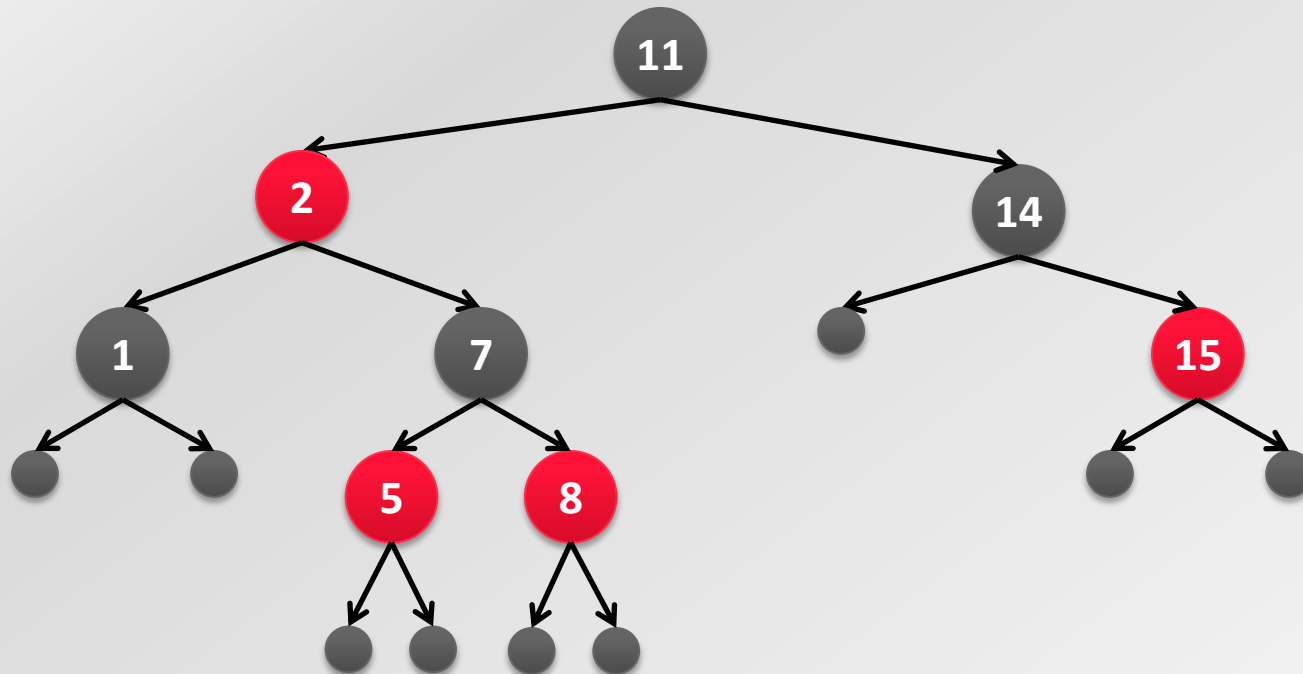
Red-Black Trees

A red-black tree is a binary search tree with the following red-black properties:

- ◆ Every node is either red or black.
- ◆ Every leaf node is black.
 - *Leaf nodes don't contain data.*
- ◆ If a node is red, then both its children are black.
 - *Implies that on any path from the root to a leaf, red nodes must not be adjacent.*
- ◆ Every simple path from a node to a descendant leaf contains the same number of black nodes.

Red-Black Trees (continued)

- ◆ Every path from the root to a leaf
 - *Has the same number of black nodes*
 - *Cannot contain two consecutive red nodes.*
- ◆ Thus, the length of the shortest and longest path can differ by a factor of at most two. The tree is **approximately balanced**.



Insertion Into Red-Black Trees

- ◆ Search to find the leaf where key should be added.
- ◆ Replace the leaf with a node containing the new key.
- ◆ Color the new node **red**.
- ◆ Add two new leaves, colored black.
- ◆ If the parent of the new node is **red**, it's a violation
 - > *can be removed by reorganizing the tree.*
 - > *May need to perform **rotations**, affecting the siblings.*
- ◆ An animation demonstrating how data is inserted into a red-black tree can be found at:
 - > <http://www.ece.uc.edu/~franco/C321/html/RedBlack/redblack.html>

Red-Black Tree Rotation

- ◆ During insert and delete operations, we may modify the tree and the result may violate the red-black properties.
- ◆ To restore the properties, we must change:
 - *The color of some of the nodes.*
 - *Their pointer structure.*
- ◆ Rotations are applied to change the pointer structure and restore tree balance.
 - *The tree can be re-balanced in $O(\log n)$ time*
- ◆ The tree remains balanced – guaranteeing a worst-case search time of $O(\log n)$.

B-Trees: Make the Search Tree “Wider and Shorter”

- ◆ A binary search tree has a branching factor of 2
 - *With each comparison we divide the search space into 2 parts, cutting the remaining search space in half.*
 - *Hence $\log_2 n$ search time – “log base two”*
- ◆ Why can't we increase the branching factor to speed up the search time?

B-Trees

- ◆ Assume t children per node
- ◆ Instead of just one key per node, they have $t-1$ keys per node (possibly kept in a hash-table).
- ◆ This creates a large branching factor.
 - *Fewer comparisons, fewer disk accesses.*
- ◆ For example:
 - *A B-tree with 10 children per node has search performance $O(\log_{10} N)$.*
 - *For 1 million data entries, 6 levels in the B-Tree, so 6 nodes need to be checked.*
 - *A BST would need to check $O(\log_2 N)$ nodes.*
- ◆ Used to implement indices in databases, to get very fast lookup performance.

Comparison: Balanced Tree Performance

Structure	Best*	Worst*	Pros/Cons
AVL	$\log N$	$\log N$	Rotations complicated to implement. Faster update performance than red-black trees.
Red-Black	$\log N$	$2\log N + 1$	Harder to implement than AVL trees.
2-3-4	$\log N$	$\log N$	Good for small data sets, but no advantage over B-tree.
B-Tree	$\log_t N$	$\log N$	Harder to implement & maintain. Minimizes disk accesses if data can't fit into memory. Larger branching factor -> better performance.
Splay Tree	$\log N$	$\log N +$	Frequently used data is closer to root - faster access. Assumes some data is used more frequently, and is worth the amortized cost of rotation.

Algorithms



- ◆ **Divide and Conquer Paradigm**
- ◆ **Searching**
- ◆ **Sorting**

Divide and Conquer Algorithms

◆ Idea:

- *Split up the problem into one or more smaller instances of the same problem*
- *Solve each smaller problem separately (with the same algorithm)*
- *Combine the results of the sub-problems to obtain a solution to the main problem*

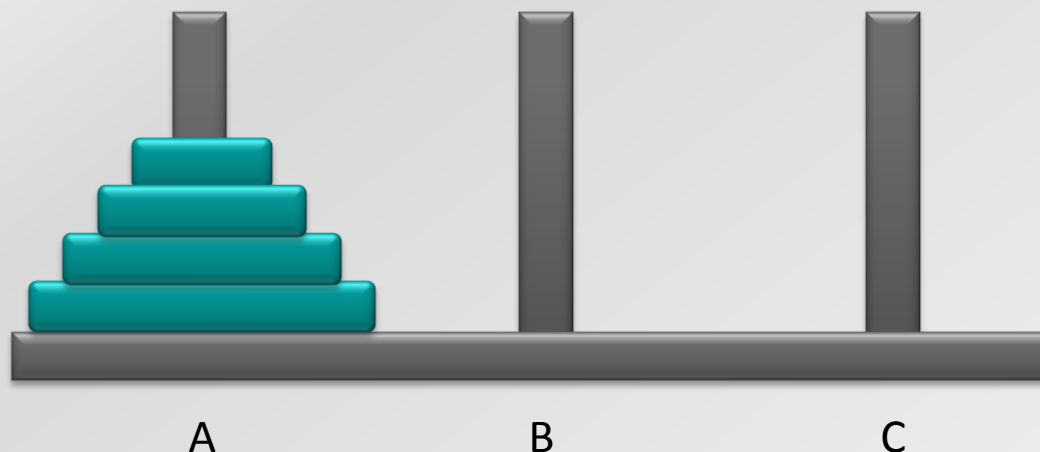
◆ Many programming languages allow the use of *recursive functions* - functions that call themselves

- *Recursion provides an elegant (though not always efficient) framework to implement divide and conquer algorithms.*

Divide and Conquer Example

Problem: (Towers of Hanoi)

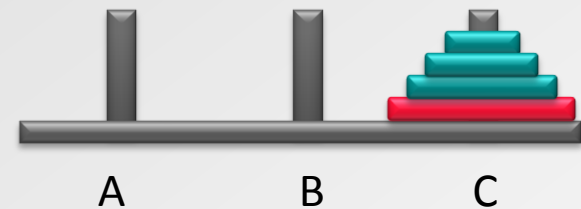
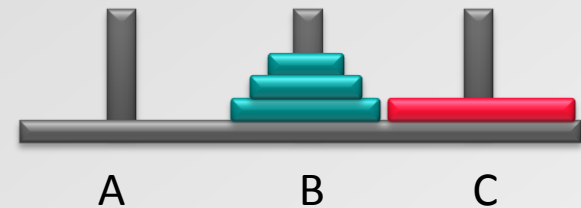
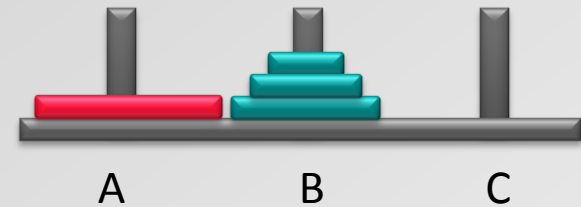
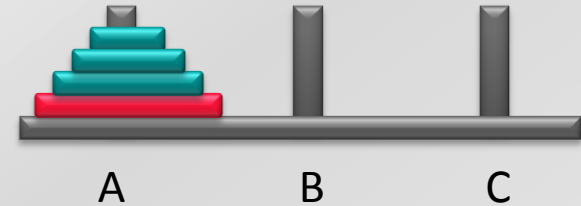
*There are three pegs **A**, **B**, and **C**. A holds **n** disks of decreasing size. Move all disks from peg A to C, one at a time, without placing a larger disk on top of a smaller disk.*



Divide and Conquer Example: Towers of Hanoi

Solution:

- > Move all but the largest disk from **A** to **B**
- > Move the largest disk from **A** to **C**
- > Move all the disks on **B** to **C**.
- > Use the same process to move disks from **A** to **B**, and from **B** to **C**.



Algorithms

- ◆ **Divide and Conquer Paradigm**



- ◆ **Searching**

- ◆ **Sorting**

Search Algorithms

- ◆ **Searching:** retrieving a specific datum from a large amount of previously stored data.
 - *Information is divided into **records**.*
 - *A **search key** is used.*
- ◆ **Elementary search methods:**
 - *Sequential search*
 - *Binary search*

Sequential Search

- ◆ Examines each element in order.
- ◆ If item found in list,
 - *average time is $O(N/2)$, where N is the list size.*
- ◆ If item not found,
 - *average time is $O(N)$.*

Binary Search

- ◆ Applies “Divide and Conquer” paradigm.
- ◆ Expects data to be sorted.
- ◆ Technique:
 - *Inspect the middle element,*
 - *If its value is greater than what we are looking for, look in the first half*
 - *Otherwise, look in the second half*
 - *Repeat until the item is found or no more elements to search*

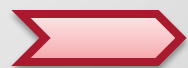
Binary Search cont.

- ◆ Works best on static data.
- ◆ Insertion is expensive.
- ◆ Complexity: average time $O(\log N)$.
- ◆ View an animated binary search [here](#).

Algorithms

- ◆ **Divide and Conquer Paradigm**

- ◆ **Searching**



- ◆ **Sorting**

Sorting Algorithms

Problem:

Given an array with n values, sort them.

◆ Elementary sorting methods:

➤ *Selection Sort*

➤ *Insertion Sort*

◆ Divide And Conquer methods:

➤ *Merge Sort*

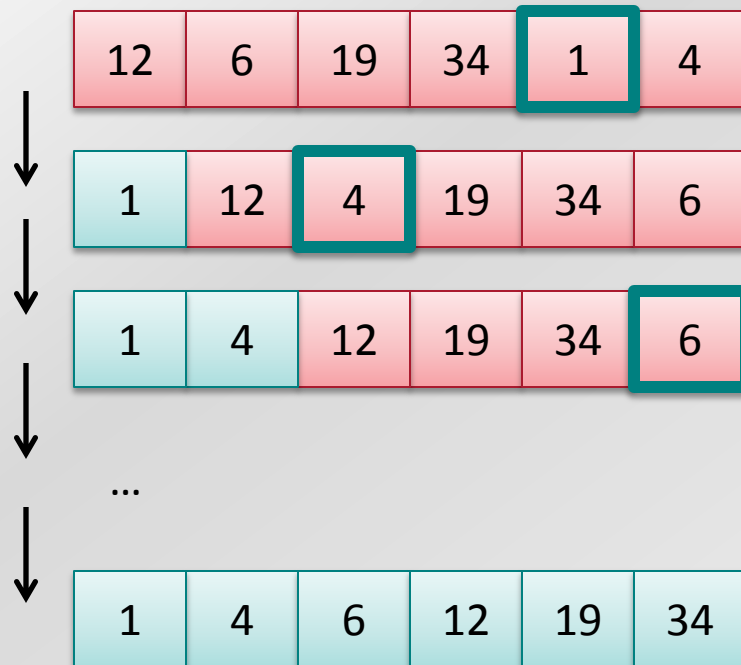
➤ *Quicksort*

Selection Sort

♦ Idea:

- *Repeatedly select the smallest value from the remaining "unsorted" values in the array.*
- *Append it to the "sorted" values in the array.*

♦ Time Complexity is $O(n^2)$



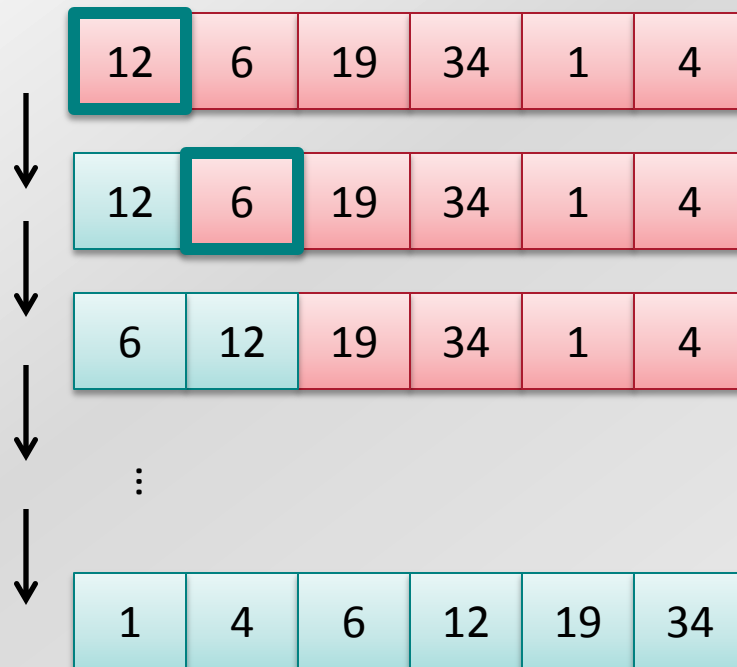
[See a selection sort animation here.](#)

Insertion Sort

♦ Idea:

- *Repeatedly pick the next value from the remaining "unsorted" values in the array.*
- *Insert it into its proper place among the "sorted" values.*

♦ Time Complexity is $O(n^2)$



Insertion sort animation [here](#).

Merge Sort

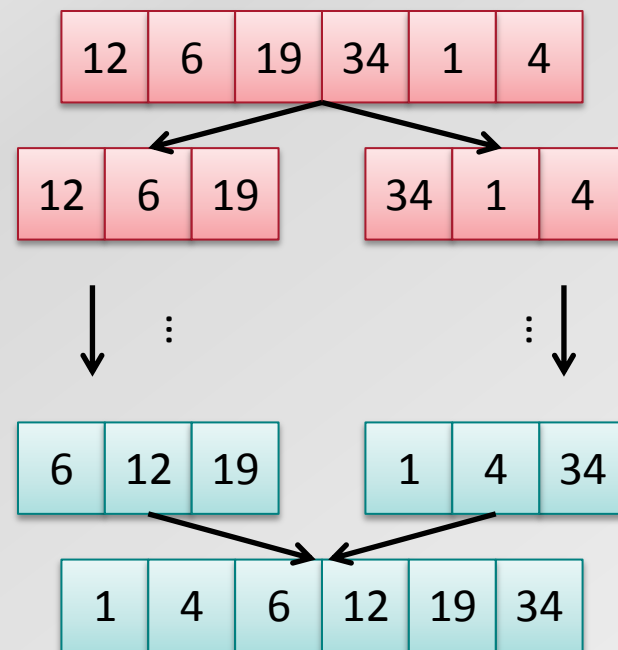
♦ Idea:

- > *Split the array into two unsorted halves.*
- > *Use merge sort to sort each half.*
- > *Merge the two sorted halves together.*

♦ Requires **$O(n \log n)$** comparisons in the worst case.

♦ Requires twice the space of the unsorted data (for the merging step).

♦ Merge sort animation [here](#).



Quicksort

♦ Idea:

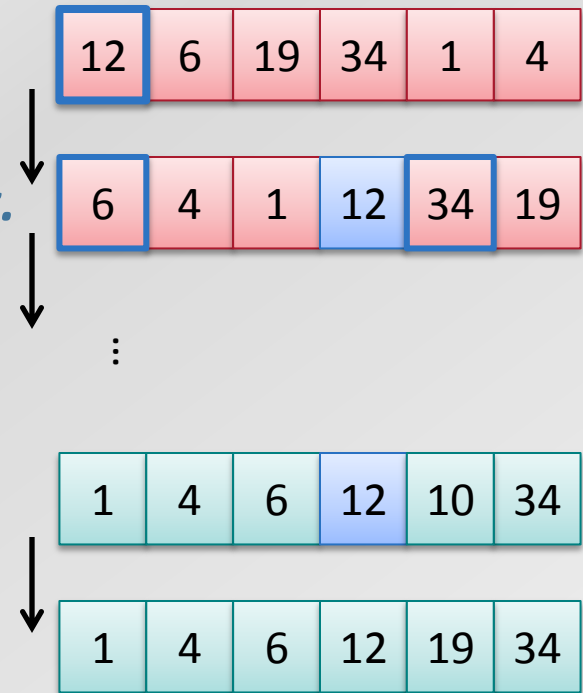
- *Pick a value from the array as a **pivot***
- *Move values smaller than the pivot to its left, move values bigger to the right.*
- *Use Quicksort to sort the left and right segments. (“Divide and conquer!”)*

♦ Requires $O(n \log n)$ comparisons in the **average case**.

♦ Requires $O(n^2)$ comparisons in the **worst case**.

♦ Merge sort **sorts in place** - it does not need any additional space.

♦ Quicksort animation [here](#).



Sorting Algorithms

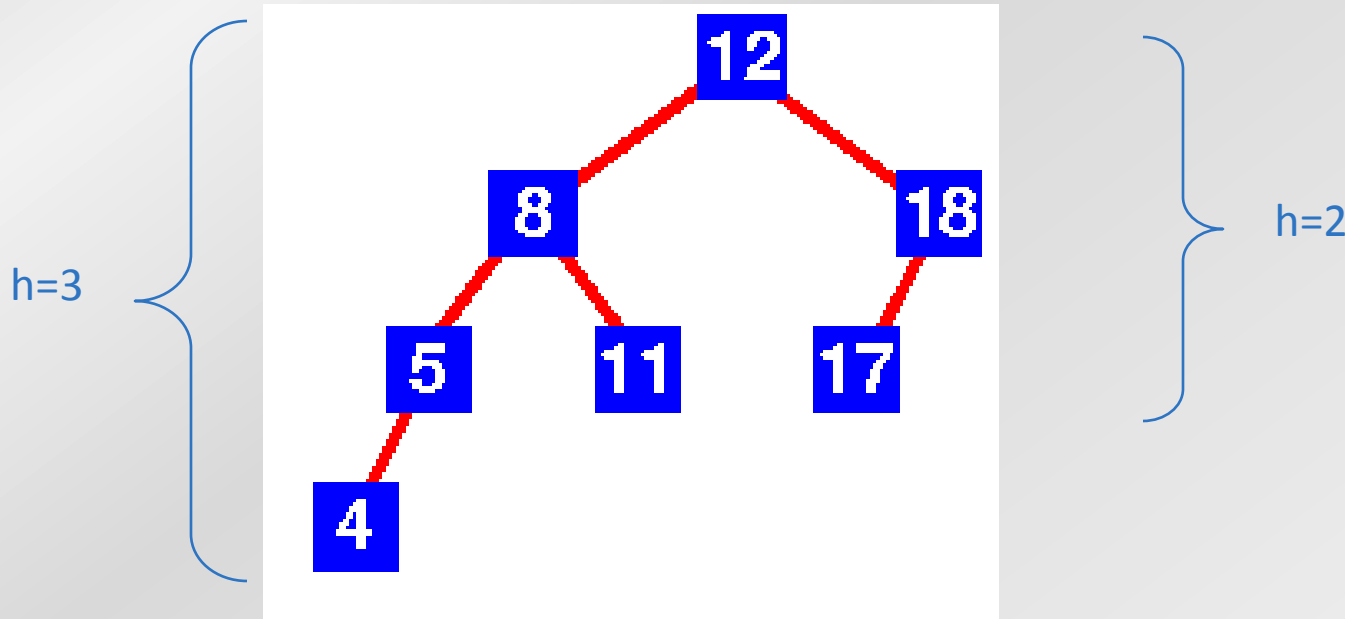
Sort	Best case	Average case	Worst case	Memory
Bubble	$O(n)$	-	$O(n^2)$	$O(1)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell	-	-	$O(n^{1.5})$	$O(1)$
Binary tree	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Radix	$O(n)$	$O(n \log n)$		
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$

Appendix

Additional Variations of Balanced Trees

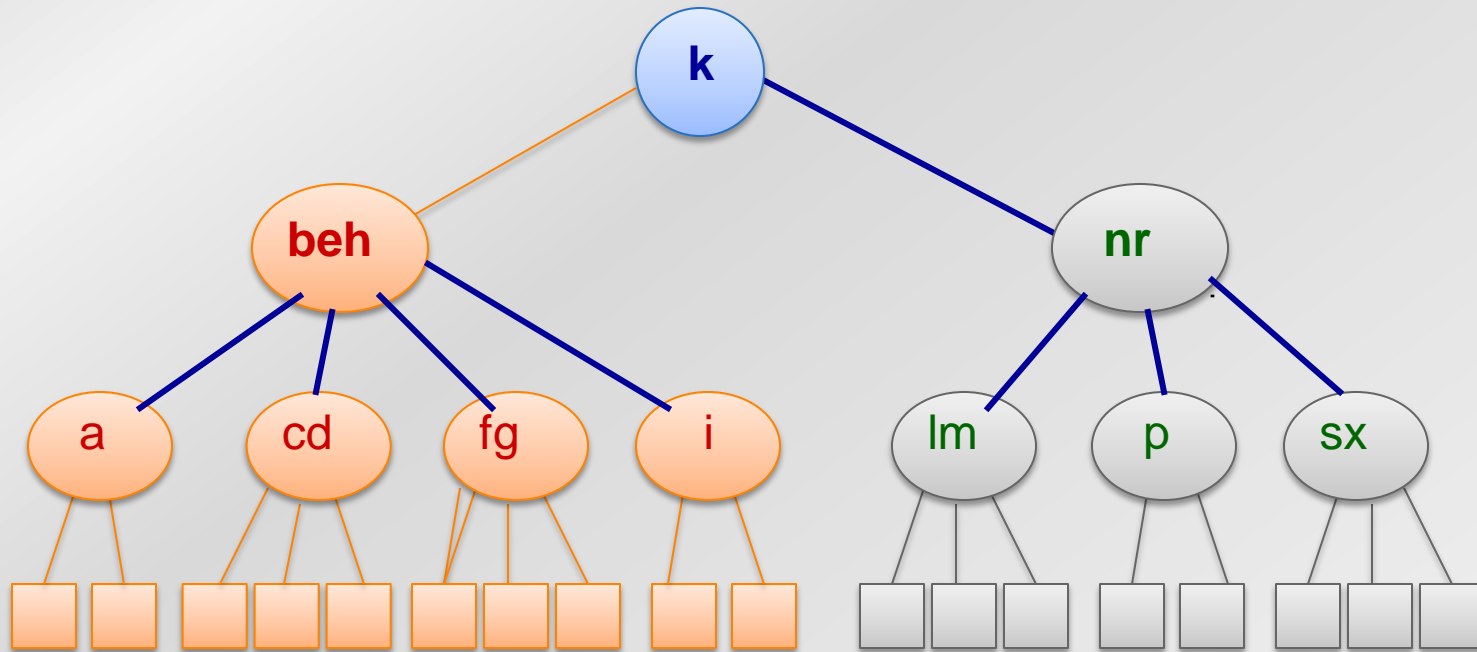
Balanced Trees: AVL trees

- ♦ AVL tree: a partially balanced binary tree where the heights of the two sub-trees of each node differ by no more than 1.



Topdown 2-3-4 Trees

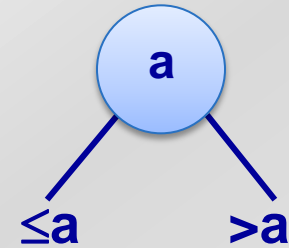
- ◆ Nodes store 1,2 or 3 keys and have 2, 3, or 4 children, respectively.
- ◆ All leaves have the same depth.



Topdown 2-3-4 Tree Nodes

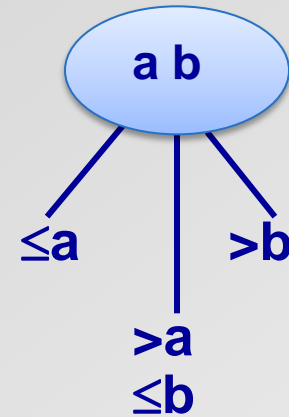
2-Node

- ◆ Same as binary node



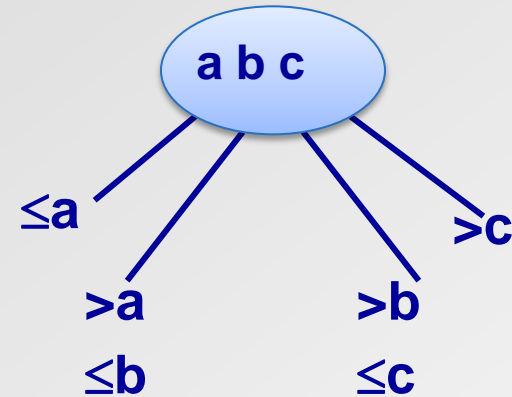
3-Node

- ◆ 2 keys, 3 links



4-Node

- ◆ 3 keys, 4 links



Insertion Algorithm on Topdown 2-3-4 Trees

- ◆ Search for the key in the tree, and split 4-nodes on the way down.
- ◆ Find the empty node where the value belongs.
 - *If it is a 2 node, convert the 2 node to a 3 node and add the new key value.*
 - *If it is a 3 node, convert it into a 4 node.*
 - *If it is a 4 node,*
 - Split the 4 node into two 2 nodes
 - Pass one key up to the next higher node, then add the new key to one of the 2 nodes, converting it to a 3 node.

Insertion Algorithm on Topdown 2-3-4 Trees

- ◆ By splitting the 4-nodes on the way down, you guarantee that:
 - *Only the level directly above the split is affected.*
 - *Levels above the split's parent are unaffected.*
 - *Nothing below the split needs to change.*

Additional Sorting Algorithms of Interest

Shellsort

- ◆ Shellsort is an improvement to insertion sort
 - *Exchanges elements that are far apart.*
 - *This decreases sorting time*
- ◆ Compares **a[j]** and **a[j-h]**
 - *On first pass **h** is big.*
 - ***h** gets smaller on each successive pass until it's 1.*
- ◆ Worst Case: **$O(n^2)$** – but can be improved
- ◆ Shellsort animation [here](#).

Radix Sort

- ◆ A multi-pass sort algorithm
- ◆ Distributes each item to a **bucket** according to part of the item's **key** (beginning with the least significant part of the key).
- ◆ After each pass, items are collected from the buckets, keeping the items in order, then redistributed according to the next most significant part of the key.
- ◆ Examples: sorting a deck of cards by suit and by A-K within each suit.

Radix Exchange Sort

- ◆ Examines the bits in the keys from left to right and manipulates the records in a manner similar to Quicksort.
- ◆ Does not work well on files containing many equal keys.

C++ Standard Template Library

C++ Standard Template Library

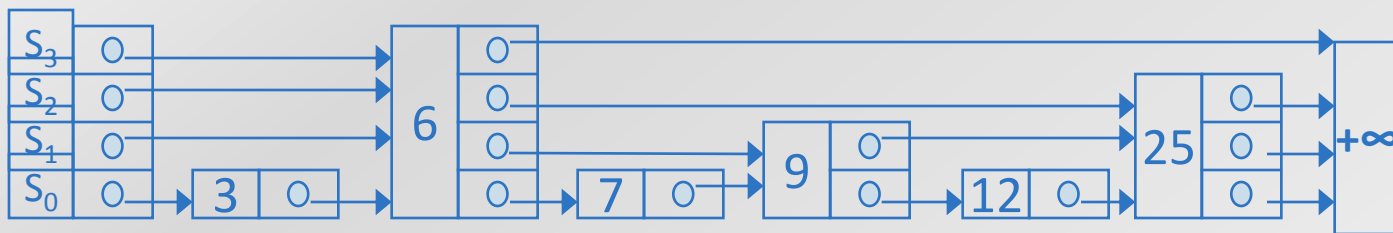
- **Containers, Algorithms, and Iterators**
 - *Map, Set, Multimap, Multiset¹*
 - *List, Vector, Deque*
 - *Stack, Queue, Priority Queue (Adaptors)*
 - *Sorting, Searching, Modifying, etc.*
- **Many implementations**

¹Implemented as balanced trees!

Another Interesting Data Structure: Skip Lists

Skip Lists

- ◆ A skip list is a collection of linked lists, S_0, \dots, S_k .
 - Although there are $k+1$ lists, the lists share nodes.
 - The key of the last element is $+\infty$ for each list.
 - The elements of each list are sorted in non-decreasing order.
 - Each list is a sub-sequence of the previous list
 - Elements of list S_i are elements of list S_{i-1}
- ◆ A node in list S_i will have an array of $i+1$ next pointers



Searching in a Skip List

To search for an element with a target key:

1. Start the search in the list with the fewest elements, S_k .
2. Search the list until you find an element whose key is larger than the target key
 - *At this point no other elements in this list can match the target*
3. Drop back to the previous node
4. Drop down to the next list
5. Repeat steps 2 through 4 until you:
 - *Find the element with a matching key, or*
 - *Find a key less than the target in S_0 or*
 - *Reach the end of the list.*

Inserting into a Skip List

◆ Demonstration

Complexity of Skip Lists

- ◆ Space used is $O(n)$
- ◆ Search, insertion, and deletion time are $O(\log n)$
- ◆ Easier to implement than B-trees and Red-Black trees
- ◆ More efficient when inserting a large number of consecutive elements.

References

- ◆ <http://www.cs.brown.edu/courses/cs016/book/> “Algorithms in C++ “ by Robert Sedgewick.
- ◆ http://www.cs.auckland.ac.nz/software/AlgAnim/ds_ToC.html
“Univ. of Auckland, N.Z. Algorithm Animations”
- ◆ <http://www.nist.gov/dads/>
- ◆ <http://www.csse.monash.edu.au/~dwa/Animations/index.html>
- ◆ <http://developer.gnome.org/doc/API/2.0/glib>
GLib Reference Manual
- ◆ <http://www2.hig.no/~algmet//animate.html>
Relevant Algorithm Animations/Visualizations (in Java)
- {BDE <GO>}
- www.cplusplus.com/reference/stl
STL Reference

Contact Information

◆ Tom Arcidiacono

Bloomberg Financials L.P.

tarcidiacono@bloomberg.net

◆ Detlef Ronneburger

Bloomberg Financials L.P.

dronneburger@bloomberg.net