CS 61B: Data Structures Lecture Notes

Week 2: Lecture 3 (1/27)

Values vs. Containers
Structured Containers - contain (0 or more) containers
IntList - Linked List of ints head, tail
Destructive - Changes the Linked List
Non-Destructive - Doesn't change the Linked List

Week 2: Lecture 5 (1/31)

Loop invariant - true at the start of a loop
Merge Sort -

Problem: Given two sorted arrays of ints, A and B, produce their merge: a sorted array containing all from A and B.

Week 3: Lecture 6 (2/3)

Multidimensional Arrays in Java
Int[][] zero = new int[3][];
Zero[0] = zero[1] = zero[2] = new int[] {0, 0, 0};
Zero[0][1] = 1;

- All zeros[i][1] will change because they point to the same array

Testing

- *Unit testing* - testing the individual units within a program, rather than the whole program
- *Integration testing* - testing of entire (integrated) set of modules - the whole program
- *Regression testing* - testing with specific goal of checking that fixes, enhancements, or other changes have not introduced faults (regressions)

Test-Driven Development

- Idea: write tests first
- Implement unit at a time, run tests, fix and refactor until it works.

Testing sort

- Corner cases - ex.) empty array, one-element, all elements the same
- Representative "middle" cases - ex.) elements reversed, elements in order, one pair of elements reversed

JUnit

- Java annotation @Test on a method tells the JUnit machinery to call that method

Week 3: Lecture 7 (2/5)

Object- Based Programming

- Function-based programs - organized primarily around functions that do things

- Object-based programs - organized around types of objects used to represent data; methods grouped by type of object

Philosophy

- We prefer a purely procedural interface where functions do everything - no outside access to the internal representation

Getter Methods

- Allow public access only through methods so that not everyone can assign to the balance field

Class Variables and Methods

- Class-wide variable means static

Multiple Constructors and Default Constructors

- Overloaded constructors - multiple constructors and can use each other
- "this" is similar to self in python

Week 3: Lecture 8 (2/7)

Overloading

- Multiple method definitions with the same name and different numbers or types of arguments

Generic Data Structures

- Any reference value can be converted to type java.lang.Object so can use Object as the "generic (reference) type"
    - Object[] things = new Object[2];
    - Things[0] = new IntList(3, null); things[1] = "Stuff";

Primitive Values?

- Use wrapper types, one for each primitive type: Byte, Long, Float, Short, Character, Double, Integer, Boolean
- Boxing: Integer Three = new Integer(3);
- Unboxing: int three = Three.intValue();

Type Hierarchies

- A container with (static) type T may contain only a (dynamic) value subtype T

Extending a Class

- B is a subclass of A
    - Class B extends A
- Can override an instance method but not a static method

Week 4: Lecture 9 (2/10) Interfaces and Abstract Classes

- Class Parent{...}
- Class child extends Parent{...}
- Child tom = new Child();
- Parent pTom = tom;
- Fields hide inherited fields of some name; static methods hide methods of the same signature.

What’s the Point?
- Define a kind of generic method
- A superclass defines a set of methods that are common to many different classes.
- Subclasses can then provide different implementations of these common methods
- All subclasses will have at least the methods listed by the super-class

Abstract Methods and Classes
- Instance method can be abstract, No body given; must be supplied in subtypes.
- Can write methods that operate on Drawables
- For (Drawable thing: thingstoDraw) thing.draw();
- Can create class Rectangle and class Oval that extends Drawable

Aside: documentation
- Can use @Override annotation for a method that will override a method in the superclass

Interfaces
- Description of the functions or variables by which two things interact.
- Can define abstract methods

Implementing interfaces
- Public class Rectangle implements Drawable{...}

Multiple Inheritance
- Can extend one class, but implement any number of interfaces.
- Class Variable implements Readable, Writeable {...}

Week 4: Lecture 10 (2/12) OOP mechanism and Class Design

Extending Supertypes, Default Implementations
- Default methods: instance methods and are used when a method of a class implementing the interface would otherwise be abstract
- Ex) default void scale (double size) {...}

Will usually be the subclass it is defined as

Specification Seen by Clients
- Clients of a module use the methods of a

Week 4: Lecture 11 (2/14) Comparable & Reader

Comparable
- Interface to describe objects that have an order String, Integer
- Can use comparable in a method max to take in integers or strings

Java Generics (I)
- Public interface Comparable<T> where T is a type

Readers
- java.io.Reader, but it is abstract
- StringReader extends Abstract Reader that implements all the methods
- Can have a method that uses the Reader r as a parameter and it will work for any reader

Lessons

- Reader interface was a specification for set of readers
- Usually client methods will specify type Reader, not a specific kind of Reader
- Clients methods are as widely applicable as possible

Week 5: Lecture 12 (2/19) Additional OOP Details, Exceptions

Parent Constructors

- One of the parent's constructors is called first
- Call to parent constructors at beginning of every one of child's constructors

Default Constructors

- Creates and calls default (parameterless constructor)

Using an Overridden Method

- Call super.*<method name>* to add to the action defined by superclass

Trick: Delegation and Wrappers

- Can have an interface which contains another Reader and delegates tasks
- Or interface Storage, class that implements Storage, and then makes Storage variables - wrapper class

Errors

- Throw exception objects
    - " throw new SomeException" - must be of Throwable type
- Try { //stuff that might throw exception} catch (Some Exception e) { //do something }
- New way for handling exceptions: catch (IllegalArg | IllegalState..) { }

Unchecked Exceptions

- Error: serious unrecoverable errors, or *RuntimeException*
- Programmer errors or errors detected by Java
- Can be thrown anywhere at any time

Checked Exceptions

- Exceptional circumstances that are not necessarily programmer errors - any other error
- Use try or report in method's declaration
- *Void myRead() throws IOException, Inter {...}*

Good Practice

- Problem may depend on caller, throw exceptions over print
- Ex. When programmer violates preconditions
-

Week 5: Lecture 13 (2/21) Packages, Access, Loose Ends

Package Mechanics

- By default, a class resides in the anonymous package
- Package declaration at the start of file

Access Modifiers

- ex.) private, public, protected
- Allow a programmer to declare which classes are need to access which declarations

Access Rules: Public

- Available anywhere
- Intentions: what clients of a package use

Access Rules: Private

- Private members are available only within the text of the same class, even for subtypes
- Intentions: part of the implementation of a class that only that class needs

Access Rules: Package Private

- Available within the same package
- Default when public or private is not specified
- Intentions: must be know to other classes that assist in implementation

Access Rules: Protected

- Package private, and available within subtypes of C1 outside package, but only if static types are subtypes of C2
- Intentions: part of implementation that subtypes may need but clients don't

What May be Controlled

- Can override a method only with one that has at least as permissive an access level

Importing

- Just means you can just List as an abbreviation for java.util.List
- You can import static members like import java.util.System.out;

Nesting Classes

- If only used in the implementation of the other or conceptually "subservient"

Inner Classes

- Can see the private variables of the enclosing class
- Non-static nested classes are called inner classes

Instanceof

- Can check dynamic type with *if (r instanceof TrReader)* but you should just use instance methods

Week 6: Lecture 14 (2/24) Integers

Integer Types and Literals

Type	**Bits**	**Signed?**	**Literals**
byte	8	Yes	Cast from int: (byte 3)
short	16	Yes	None. Cast from int: (short) 4096
char	16	no	'a' // (char) 97
Int	32	Yes	123 0x3f, // Hexadecimal
long	64	Yes	123 0100

- "N" bits means that there are 2^N integers in the domain of the type
- Signed, range of values is -2^(N-1) .. 2^(N-1) -1
- Unsigned, only non-negative numbers, and range is 0... 2^(n-1)

Overflow

- Java defines the result on integer types to "wrap around" -- modular arithmetic

Week 6: Lecture 15 (2/26) Integers

Negative numbers

- For signed, starts with 1
- -1 represented as 11111111, when added with 1, 00000001, it becomes 1 | 0000000 which will equal 1
- Unsigned will have the same except 111111111 is 2^16 -1

Conversion

- Will convert from one type to another if it makes sense and no information is lost from the value
- Long => Int might lose information

Promotion

- Promote operands as needed, if any operand is a long, promote both to long

Bit twiddling

- Operators and their uses
    - Mask: num & num as and
    - Set: num | num as or
    - Flip: num "^" num as unequal, where if 1 and 1 = 0
    - Flip all: num "~", not operator
- Shifting
    - Left: "<<"Shift bits left and everything that is past, falls off
    - Arithmetic Right: ">>"keeps sign bit and shifts right
    - Logical Right: ">>>" shifts right and starts with 0
    - (-1) >>> 29 = 7

Week 6: Lecture 16 (2/28) Complexity

Operational costs, development costs, maintenance costs, costs of failure
Keep It Simple
Cost Measures (Time)

- Wall-clock, execution time
    - Easy to measure, time it takes on a machine, complier
- Dynamic statement counts
    - # of times statements executed
    - Don't know actual time
- Symbolic execution times
    - Formulas for execution times as functions of input size
    - Don't know actual time

Asymptotic Cost

- Symbolic execution lets you see shape of cost function
- Interested in asymptotic behavior as input size becomes very large

Order Notation
Runtime

- Only largest degree of worst runtime counts
- Simplification
    - 1) Consider only worst case
    - 2) Pick representative proxy for overall runtime - "cost model"
    - 3) Ignore lower order terms
    - 4) Ignore multiplicative constants

Big Theta (ø(f(N))

- Order of growth is f(N)

Big O (O(f(N))

- Order of growth is less than or equal to f(N)

Big Omega (**Ω**(f(N))

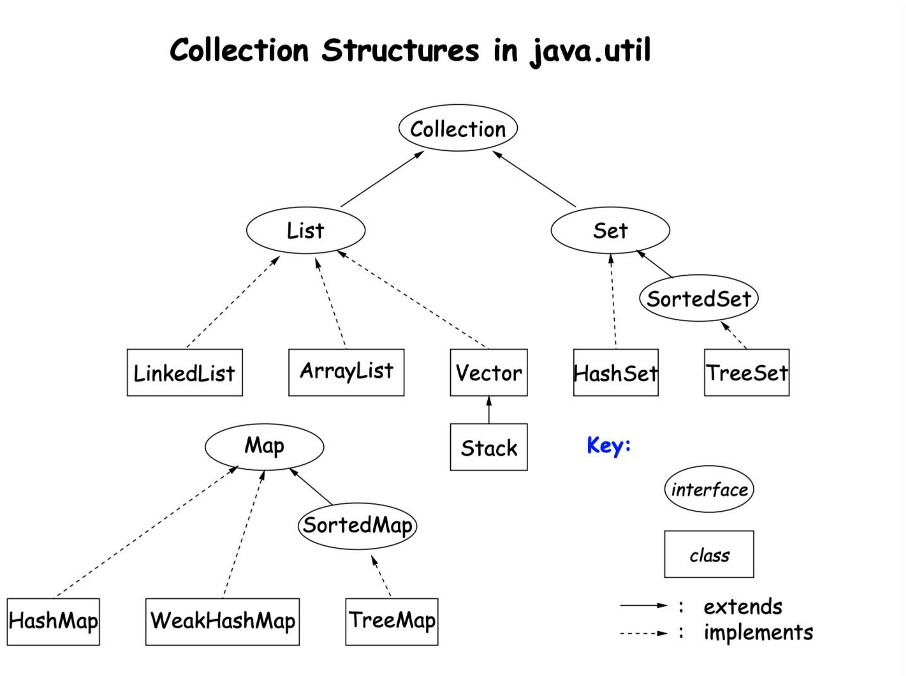
- Specify bounding from below

Week 7: Lecture 17 (3/2) Collections Amortization

Data Types in the Abstract

- Interfaces
    - Collections : General collections of items
    - List : Indexed sequences with duplication
    - Set, Sorted Set : Collections without duplication
    - Map, SortedMap : Dictionaries (key -> value)
- Concrete Classes

- LinkedList, ArrayList, Hashset, TreeSet



List interface
- New methods: indexOf, get(i), add(index)

ArrayList
- Made up of list, once you fill it up, it will create a list that is 2 times the size that was filled so that it is O(1) when you add another element

Amortized Time
- Given g(i) << f(i) and ci \in O(f(i)) for differing sequence and ai \in O(g(i))
- If \sum a > \sum c for all k, we can say the operations run in O(g(i)) amortized time
- Having one long runtime for making new array and constant time for adding into array

Amortized Time: Potential Method
- Associate a potential that keeps track of saved up time from cheap operations so that we can spend on later expensive ones, like a bank account

Week 7: Lecture 18 (3/4) Assorted Topics

Views
- Alternative presentation of an existing object, ex. Sublist method of list takes

Maps
- Methods: get(Object key), put(Key key, Value value)
- TreeMaps: Ordered maps

AbstractList<Item> implements List<Item>
- Boolean contains(Object x),

AListIterator
- Implements hasnext(), next() for liste

Arrays and Links

- Array:
    - Advantages: compact, fast random accessing (indexing)
    - Disadvantages: insertion, deletion can be slow
- Linked list
    - Advantages: insertion, deletion fast once position found
    - Disadvantages: space, random access slow

Arrays

- Problem is insertion/deletion in the middle of a list, must move elements
- Can use circular buffering where there is space in the middle between last and first to grow at either end

Linking

- Java LinkedList, can use a listIterator() object over it

Sentinels

- Dummy object that just links, fixed object to point

Specialization

- Stack: add and delete from one end (LIFO)
- Queue: Add at end, delete from front (FIFO)
- Dequeue: Add or delete at either end.

Design Choices: Extension, Delegation, Adaptation

Week 8: Pattern Matching (RegEx)

Character class ([0-9abd-qs-z])

- Any of the single characters

Wildcard (.)

- Period can match any character

Compliment, Not ([^abe])

- Matches any single character other than those listed

Character Class shortcut (\s, \d)

- Whitespace, [0-9], need to use "\\d" in order to use \ key

Repetitions (*, +, ?)

- P* is "0 or more repetitions of P"
- P+ is "1 or more Ps"
- P? Is "0 or 1 Ps"

Or (P |Q)

- Either "a" or "b"

Group ((P))

- Subpattern you can retrieve later

Escape (\?, *, \., \+)

- Need to use two-character escape sequences to match character after backslash (\\?)

Week 8: Lecture 20 (3/9) Trees

Trees Definition

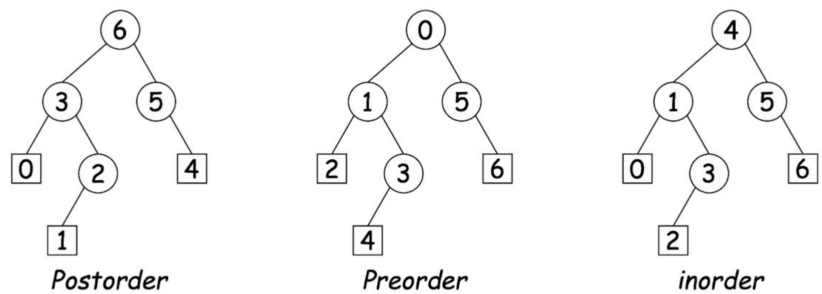
- Label value and branches

Characteristics

- **Root**: non-empty node with no parent in that tree (all nodes are roots of subtree)
- Order, Arity, degree: of a node is number of children
- **Leaf** has no children
- **Height** is largest distance to a leaf
- **Depth**: distance to the root of that tree

Fundamental Operation: Traversal

- Traversing a tree is enumerating some of its nodes
- When nodes are enumerated, they are visited
- **Preorder:** visit node, traverse its children
- **Postorder:** traverse children, visit node
- **Inorder:** traverse first child, visit node, traverse second child (binary tree)



Visitor Pattern

- Consumer<AType> interface that has method accept.

Depth-First Traversals

- Add to a stack (LIFO)

Level-Order (Breadth-First) Traversal

- Traverse all nodes at depth 0, then depth 1,... LIFO to FIFO

Iterator for Trees

- PreorderTreeIterator<Label>, using preorder implement Iterator and *hasNext, next* method

Week 8: Lecture 21 (3/11) Tree Searching

Divide and Conquer

- Preferable to have criteria for dividing data to be searched into pieces recursively

Binary Search Trees

- All nodes in left subtree have smaller keys
- All nodes in the right subtree of node have larger keys

Binary Tree Insertion

- Insertion - Search then input tree where it should be
- Time proportional to height

Deletion

- No child: Replace tree with null if it is leaf,

- One child: Child could replace the parent
- More than one: Find smallest number in right side and replace it

Quadtree

- Index information about 2D Locations so that items can be retrieved by position
- Looking for point (x’,y’) and narrow down which four subtrees

Week 8: Lecture 22 (3/13) Game Trees

Searching by “Generate and Test”

- Can enumerate all possible candidates and test all possibilities in turn

Backtracking search (knightMove)

- Board with where you have visited, and path
- For all possible moves, if you haven’t visited, add a path, then findPath to end, if you can find then return true, else erase that visited square, remove path

Finding the best move

- Assign a heuristic (guess) value to each possible move and pick highest
    - Ex. Chess sum of pieces in (Queen = 9, Rook = 5)
- Move might give us more pieces but set up devastating response from oppoent

Game Trees

- Space as possible continuations of the game as a tree
- Each node is a position, each edge a move
- Levels: my move, opponent’s move, my move, opponent’s move
- I choose child with max value, opponent chooses min value

Alpha-Beta Pruning

- **Prune** the tree as we search it, the opponent will not choose a move, and I would never choose to move here, sends down values that are acceptable and discontinues search if it is not in range

Cutting off the Search

- If you can traverse to bottom win/loss, but game trees tend to be either infinite or impossibly large
- Choose a maximum depth

Overall Search Algorithm

- Search for a move to be optimal in one direction or other
- Search exhaustive down to a particular depth in a game tree
- Pass å, ß,
    - High player does not care about a position further if greater than alpha
    - Minimizing player won’t explore positions whose value is less than maximizing player
- Maximizing player will findMax

Pseudocode

- Look at each legal move
- Try making move
- Find one with best heuristic estimate

- If beta <= alpha: break;

Week 9: Lecture 23 (3/16) Priority Queues, Range Queries

Priority Queue:
- "add" "find largest" "remove largest"

Heap
- Max-heap: binary tree that enforces Heap Property
    - Labels of both children of each node are less than node's label
- Node at top has largest label, smallest nodes are anywhere at the bottom
- Insertion and deletion take log N worst time
- Min-heap: same but min value at the root and children larger values

Heap Insert/Delete
- Insert: Find node below greater than current label, swap with one of child nodes
- Remove largest: move bottommost, rightmost to top and reheap down as needed, swapping
- Can use heaps in Arrays

Ranges
- BST can look for range of values

Time for Range Queries
- O(h + M) : *h* is height of tree, *M* is data items in range

Ordered Sets and Range Queries in Java
- SortedSet supports range queries of set
- S.headSet(U): subset S that is < U
- S.tailSet(L): subset that is >= L
- S.subSet(L, U): subset that is >=L, < U

TreeSet
- TreeSet<T> needs to be a Comparable

BSTSet
- subset1.subSet("a", "d")
    - Always pointer to BST, (top node) plus bounds

Week 9: Lecture 24 (3/18) Hashing

Simple Search
- Linear Search is OK for small data sets but bad for large
    - Can put into bucket of constant size and would be constant time

Hash functions
- Convert key to bucket number: a hash function
- External Chaining: Each bucket has list of data items, load factor: average N(items) /M(buckets) = L
- Avoid collisions: keys that "hash" to equal values

Filling the Table
- Resize table when load factor gets higher than some limit
- Rehash all items and get constant amortized time

Hash Functions: Strings

- Java uses: h(s) = s0* 31^(n-1) + s1 *31^(n-2) + ... + s_n-1
- Others are similar

What Java Provides

- Class Object has hashCode() returns identity has function

Monotonic Hash Functions

- Key k1 > k2 then h(k) > h(k) items are time-stamped records

Perfect hashing

- Hash every key to different value: perfect hashing, keys fixed

Characteristics

- Add, lookup, deletion takes Theta(1) time, bad for range queries

Here, N is #items, k is #answers to query.

Function	Unordered List	Sorted Array	Bushy Search Tree	"Good" Hash Table	Heap
find	$\Theta(N)$	$\Theta(\lg N)$	$\Theta(\lg N)$	$\Theta(1)$	$\Theta(N)$
add (amortized)	$\Theta(1)$	$\Theta(N)$	$\Theta(\lg N)$	$\Theta(1)$	$\Theta(\lg N)$
range query	$\Theta(N)$	$\Theta(k + \lg N)$	$\Theta(k + \lg N)$	$\Theta(N)$	$\Theta(N)$
find largest	$\Theta(N)$	$\Theta(1)$	$\Theta(\lg N)$	$\Theta(N)$	$\Theta(1)$
remove largest	$\Theta(N)$	$\Theta(1)$	$\Theta(\lg N)$	$\Theta(N)$	$\Theta(\lg N)$

Week 9: Lecture 25 (3/20) Java Generics

Basic Parameterization

- Public class ArrayList<Item> implements List<Item> {
- Item, Key, Value are *formal type parameters* whose values get substituted for other occurrences of Item, Key or Value

Parameters on Methods

- Figures T by looking at the type of x, type inference

Wildcards

- You don't care what type a parameter is
    - Static int frequency(Collection<?> c, Object x) {...}

Subtyping

- List<String> LS = new ArrayList<String>();
- List<Object> LObj = LS;
- In general, for T1<X> subtype of T2<Y>, must have X = Y and T1 subtype of T2

Type Bounds

- To ensure that a particular type parameter is replaced by a subtype,
- Class NumericSet<T extends Number> extends HashSet<T> {
- Requires all type parameters to NumericSet must be subtypes of Number
- Static <T> void fill(List<? Super T> L,) {..

- Must be Q where T is a subtype of Q

Week 11: Lecture 26 (3/30) Sorting

Purposes of Sorting

- Sorting supports searching (binary search)
- Sorting Algorithm permutes a sequence to bring them to order
    - Total: x subset y or y subset x for all x, y
    - Reflexive: x subset x
    - Antisymmetric: x subset y and y subset x iff x = y
    - Transitive: x subset y and y subset z implies x subset z
- Sort that does not change the relative order of equivalent entries is stable

Classifications

- Internal sorts: keep all data in primary memory
- External sorts: process large amount of data in batches, keeping what doesn't fit in secondary storage
- Comparison-based: only things we know about keys is their order
- Radix sorting: uses more information about key structure
- Insertion sorting: repeatedly inserting items at their appropriate positions in the sorted sequence being constructed
- Selection Sorting: repeatedly selecting the next larger item in order and adding it to the end of the sorted sequence being constructed

Sorting Arrays of Primitive Types

- In java.util.Arrays
- Static void sort(P[] arr)
- Static void sort(P[] arr, int first, int end)
- parallelSort

Sorting Arrays of Reference Types

- If reference type C has natural order (implement java.lang.Comparable)
- Can use sort with any R[] arr and Comparator< ? super R> comp

Sorting Lists

- java.util.Collections has two methods
- Static sort (List<C> lst)

Examples

- Sort X into reverse order sort(X, (String x, String y) -> {return y.compareTo(x);});
    - sort (X, Collections.reverseOrder())
- Sort X[10] ,... X[100] in X
    - Sort (X, 10 , 101)

Insertion Sort

- Starting with empty sequence of outputs
- Add each item from input, inserting into output sequence at right point
- Theta(N^2)
- Theta(N) is already sorted, good for any nearly sorted data, # of inversions is measure of unsortedness

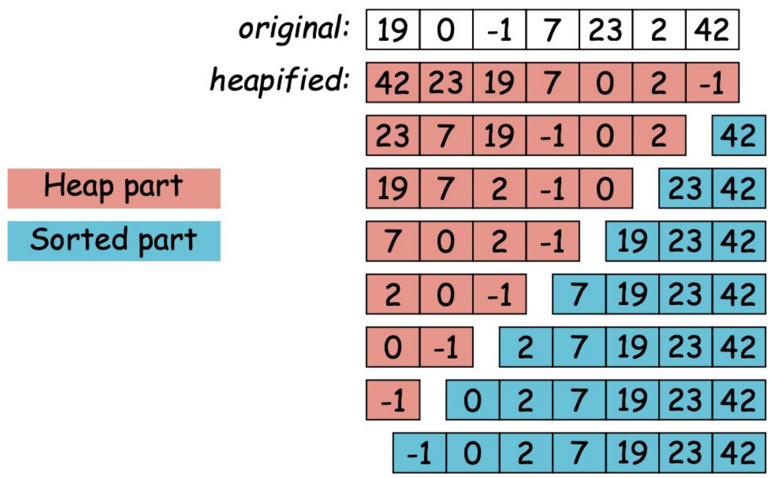
Shell's sort

- First sort distant elements: 2^k - 1, Sort is Theta(N^(3/2))

Week 11: Lecture 28 (4/3) Sorting

Sorting by Selection: Heapsort

- Idea: Keep Selecting smallest (or largest element)
- O(N log N) time



Merge Sorting

- Idea: Divide data in 2 equal parts; recursively sort halves; merge results
- Theta(N log N)
- External Sorting: break data into small chucks and sort, then merge

Quicksort: Speed through Probability

- Idea: Partition data into pieces: everything > a pivot value at the high end and everything <= on the low end
- Stop when pieces are small enough and do insertion sort because insertion has low constant factors

- In this example, we continue until pieces are size ≤ 4.
- Pivots for next step are starred. Arrange to move pivot to dividing line each time.
- Last step is insertion sort.

16	10	13	18	-4	-7	12	-5	19	15	0	22	29	34	-1*
-4	-5	-7	-1	18	13	12	10	19	15	0	22	29	34	16*
-4	-5	-7	-1	15	13	12*	10	0	16	19*	22	29	34	18
-4	-5	-7	-1	10	0	12	15	13	16	18	19	29	34	22

- Now everything is "close to" right, so just do insertion sort:

-7	-5	-4	-1	0	10	12	13	15	16	18	19	22	29	34

-
- If pivots are good: Theta (N log N)
- Bad pivots: Theta(N^2)

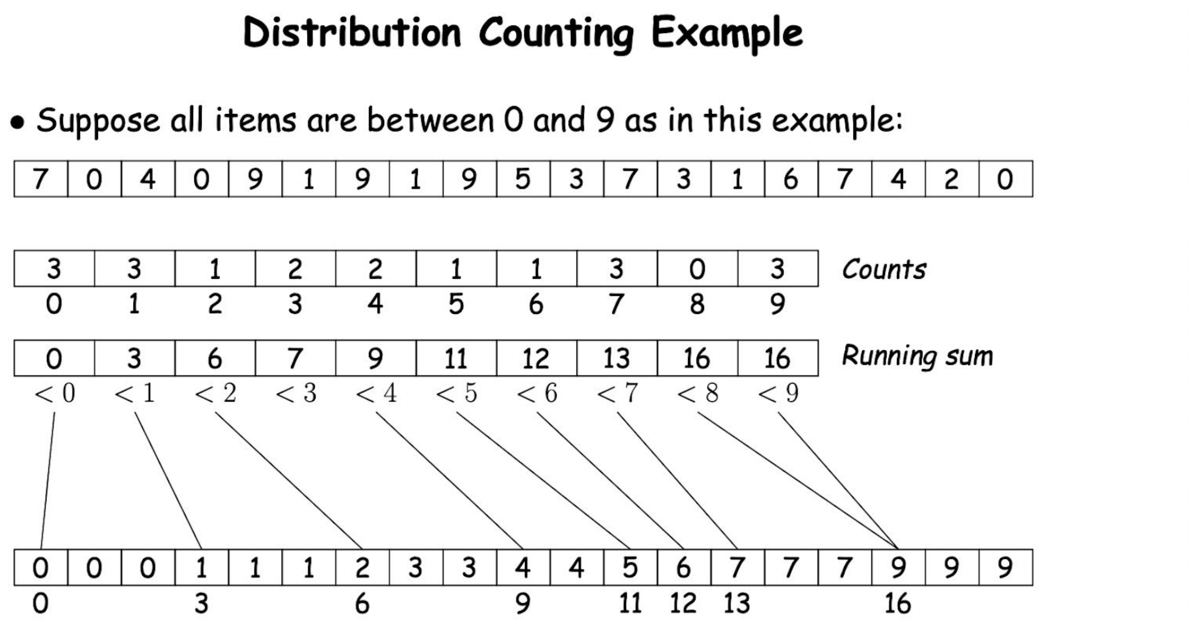
Quick Selection
- Find kth smallest element in data
- Sort for time Theta(N log N) and find k smallest Theta(N)
- Quick selection: Can do Theta(N) time for similar to quick sort with pivot
    - If m = k, p is answer
    - M > k, select kth from left half,
    - m< k select k-m-1 from right half
    - Worst case: Theta(N^2)

Week 12: Lecture 29 (4/6) Sorting

Better than N log N?
- N! Possible ways the input data can be scrambled
- Need to move N! Data and test if statements
- Worst case tests to sort N items is Omega(N log N)

Beyond Comparison: Distribution
- Put integers into N buckets, at most k keys per bucket, use insertion
- Putting in buckets takes time Theta(N) and insertion takes Theta(kN), sorting time in Omega(N)

Distribution Counting
- Mp = items with value < p, jth item with p must be $Mp + j
- With N items in the range 0.. M - 1, gives Theta(M + N)
-



Radix Sort

- Sort keys one character at a time
- Takes Theta(B) where B is total size of the key data

Search Trees

- With balance, N insertions in time logN each, plug Theta(N) to traverse, give Theta(N log N)

Summary

Sort Alg	Time Complexity	
Insertion Sort	Theta(kN) comparisons and moves	K is maximum amount data is displaced from final position - Good for small datasets or almost ordered data sets
Quick Sort	Theta(N log N), Worst: O(N^2)	Good constant factor is data not pathological
Merge Sort	Theta(N log N)	Good for external sorting
Heap Sort	Theta(N log N)	Treesort with guaranteed balance
Radix Sort	Theta(B) (# of bytes)	Good for external sorting

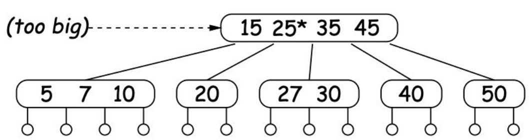
Week 12: Lecture 30 (4/8) Balanced Search Structures (B tree & LLRB)

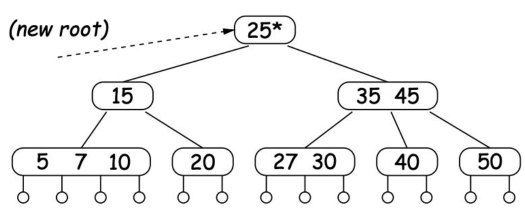
Balanced Search: The Problem

- Search trees have fast insertion/deletion and support range queries, sorting
- “Stringy” trees perform like linked lists, need to be bushy

Direct Approach: B-Trees

- Order M B-tree is M-ary search tree, M > 2
- Keys are sorted in each node
- All keys in left subtrees of a key < key, all to right are > key
- Each node has from ceil(M/2) to M children and one key “between” each two children
- Root has from 2 to M children
- Insertion: add above bottom; split overfull nodes as needed, moving one key up to parent



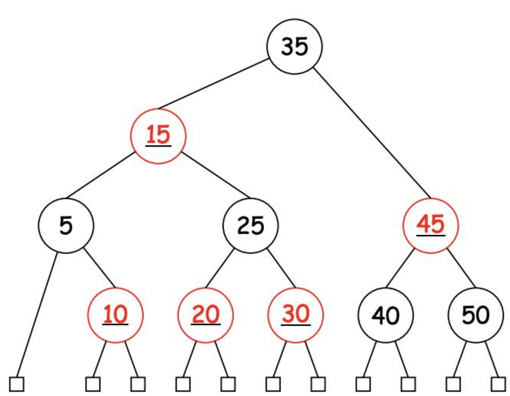


-
- (2, 3) trees have 2 elements per node and 3 children
- (2, 3, 4) or (2, 4) trees have 3 elements per node and 4 children

Red-Black Tree

- BST where Searching is always O(log N)
- When items are inserted or deleted, tree is rotated and recolored to restore balance

1. Root is black
2. Every leaf node has no data and is black
3. Every leaf has same number of black ancestors
4. Every internal node has two children
5. Every red node has two black children



- Can have left-leaning red-black trees to make one-to-one relationship between (2,4) trees and RB trees

Fixing Up the Tree

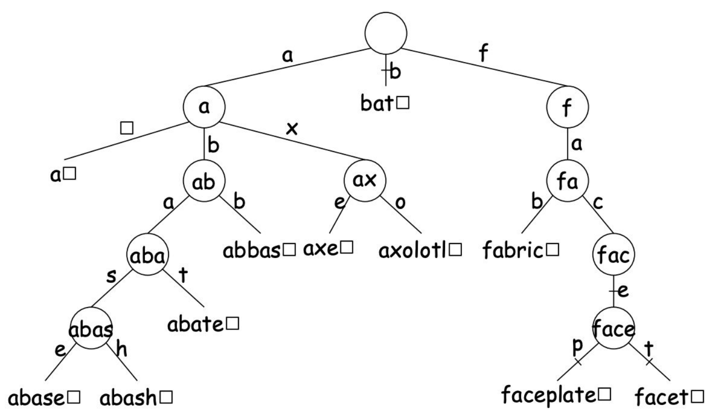
- After inserting, fix up the tree

1. Convert right-leaning trees to left-leaning
    a. (right().isRed() && left().isBlack)? rotateLeft()
2. Rotate linked red nodes into normal 4 node
    a. (left().isRed() && left().left().isRed())? tree.rotateRight()
3. Break up 4 nodes into 3 nodes or 2 nodes
    a. (left().isRed() && right().isRed()) colorFlip(tree)
4. Turn root black after fixups

Week 12: Lecture 31 (4/10) Balanced Search Structures (Trie)

Trie

- Each internal node corresponds to a possible prefix



A Side-Trip: Scrunching

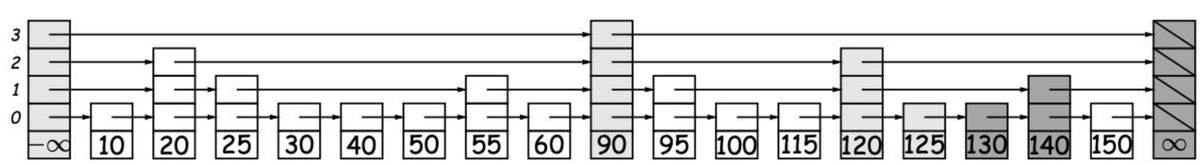
- Can have array indexed by character for internal nodes
- O(L) performance, L length of search key
- Problem: arrays are sparsely populated by non-null values -- waste of space

Scrunching

- Put arrays on top of each other use extra markers to tell which entries belong to which array
- Use arrays for the first few levels, then use linked lists for node's children

Probabilistic Balancing: Skip Lists

- Search tree in which we choose to put the keys at "random" heights
- Start at top layer on left, search until next step would overshoot, then go down one layer and repeat



-

Summary

- B-trees, Red-Black Trees:
    - Theta(log N) performance for searches, insertions, deletions
- Tries
    - Theta(B) for searches, insertions, and deletions, where B is length of key being processed
- Skip Lists
    - Probably Theta(log N) for searches, insertions, deletions
    - Probabilistic balance, randomized data structures

Week 13: Lecture 32 (4/13) Git Internals

Git: Case Study in System and Data Structure Design
- Stores snapshots of the files and directory structure of a project
- Distributed - there are many copies of a given repo

History
- Developed by linus Torvalds

Major User-Level Features
- Abstraction of versions called commits
- Graph structure reflects ancestry
- Each commit has
    - Directory tree of files
    - Information about who committed and when
    - Log message
    - Pointers to commit from which the commit was derived

Conceptual Structure
- Blobs: hold contents of files
- Trees: directory structures of files
- Commits: references to rees and information
- Tags: References to commits or identify releases, versions

User-Level Features
- Each commit has a unique name that identifies it to all versions
- Repositories can transmit collections of versions to each other
- Transmits only repos that are in one but not the other
- Repos maintain named branches which identify particular commits that are updates to keep track of the most recent commits in various lines of development

Internals
- Data of repo is stored in various objects corresponding to files, tree, and commits
- To save space, data in files is compressed
- Git can garbage-collect the objects to save space

Pointer Problem
- Objects in Git are files,
- Want to transmit objects from one repo to another with different contents, how to transmit pointers?

Content-Addressable File System

- Name objects universal, then can use names as pointers
- Uniquely identified by content, try to hash contents

Cryptographic Hash Function

- Very difficult to have a collision
- Preimage resistance: should be computationally infeasible to find a message m given h = f(m)
- Second pre-image resistance: given m_1, infeasible to find m_2 ≠m_1 such that f(m_1) = f(m_2)
- Collision resistance: difficult to find any two messages m_1 ≠ m_2 such that f(m_1) = f(m_2)

SHA1

- Git uses SHA1 (Secure Hash Function 1)
- 160 bit hash codes of contents in hex

Week 13: Lecture 33 (4/15) Graph Structures

Terminology

- Graph has set of nodes (V) and edges (E)
- Directed edges - If the edges have an order, making directed graph
- Cycle - path without repeated edges from a node back to itself
- Cyclic - if it has a cycle, else acyclic
- Directed Acyclic Graph - DAG

Trees are graphs

- Connected - a path between every pair of nodes, reachable
- Rooted - iff connected, and every node except root has one parent
- Free tree - connected, acyclic, undirected graph, free to pick the root

Representation

- Edge list Representation: each node has a list of successors
- Edge Sets: Collection of all edges ({(1, 2), (1, 3), (2, 3)})
- Adjacency matrix: Connections with matrix entry with columns for all verticies and 0, 1 representing connection

Traversing a Graph

- Algorithms on graphs depend on traversing all or some nodes
- Can’t use recursion because of cycles, want to visit nodes constant # of times

Recursive Depth-First Traversal of a Graph

- “Bread-crumb” method - mark nodes as we traverse them and don’t traverse previously marked nodes
- Preorder - mark then visit edges of a node
- Postorder - visited edges then mark node
- If not marked then mark and visit

```
void preorderTraverse(Graph G, Node v)
{
   if (v is unmarked) {
     mark(v);
     visit v;
     for (Edge(v, w) ∈ G)
       traverse(G, w);
   }
}
```

```
void postorderTraverse(Graph G, Node v)
{
   if (v is unmarked) {
     mark(v);
     for (Edge(v, w) ∈ G)
       traverse(G, w);
     visit v;
   }
}
```

-

Dijkstra's Algorithm
- Problem: Given a graph with non-negative edge weights, find shortest path from source node to all nodes
- Visit nodes in order of best known distance from source, visit all nodes and relax all edges
- For each node, keep estimated distance from s and of preceding node in shortest path from s
- Queue called fringe, with starting vertex s and mark vertex
- Implementation:
    - Fringe
    - While (!fringe.isEmpty()) {
        - Remove vertex v from fringe
        - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + weight(v, n)

<u>Week 13: Lecture 34 (4/17) A*, Minimum spanning trees</u>

Point-to-Point Shortest Path
- Dijkstra's algorithm gives shortest paths from vertex to all others in a graph

A* Search
- Wanted a path from a source to a desired vertex
- Use a heuristic guess, h(V) of a length of a path from vertex V to target V
- Order by sum of distance plus heuristic estimate of remaining distance
- Look at places that are reachable from places where we already know the shortest path

Admissible Heuristics for A* Search
- Admissible (h(C)) - must never overestimate minimum path distance

Consistency
- h(A) < h(B) + d(A, B)
- Consistent heuristics are admissible

Summary
- Dijkstra's algorithm - finds shortest-path tree to all other nodes
    - Time Complexity Theta(V log V + E log V) = (V + E) log V
- A* search - finds shortest path to a particular target node
    - Stops when target found, obey admissible and Consistency
    - Order by h(V) = d(v, target)

Minimum Spanning Trees

- Problem: Given a set of places and distances, find a set of connecting roads of minimum total length that allows travel between any two
- Will be a tree, since it is acyclic and connected
- There is no source for MST

Prim's Algorithm

- Idea
    - Start from arbitrary start node
    - Add shortest edge that has one node inside MST
    - Repeat until V-1 edges
- Implementation
    - Distance of source = 0, all others infty, insert into fringe
    - While (!fringe.isEmpty())
        - Int v = fringe.delMin()
        - scan(G, v)
    - scan(G, V)
        - Mark = true
        - For edge in G adjacent
            - If weight is smaller than distance set
                - Set dist and edge to
                - Change priority
- Runtime
    - O(V log V + V * log V + E * log V)

https://youtu.be/6uq0cQZOyoY

Kruskal's Algorithm

- Idea
    - Consider edges in order of increasing weight, add unless cycle is created
- Implementation
    - Use priority queue of fringe
    - MST = {} \\ list of edges
    - For each edge(v, w) in increasing order of weight
        - If (v, w) connects two different subtrees
            - Add (v, w) to MST
            - Combine the two subtrees into one
    - Need to find be able to for each node
        - Find which group it is in
            - When you do find operation, compress the path to the root so it directly under
        - Be able to combine two groups
            - To combine, point one root to the other

- Represent a set of nodes by one arbitrary node, let every node point to that node

https://youtu.be/ggLyKfBTABo

Week 14: Lecture 35 (4/20) Pseudo-Random Sequences

Pseudo-random Sequences

- Need a sequence that is hard or impractical to predict
- Linear Congruential Method
    - X0 = arbitrary seed
    - X_i = (aX_{i-1} + c) mod m, i > 0
    - Want a, c, m with no common factors
    - Period of m with potency (dependencies among X_i)

What Can Go Wrong?

- Short period, every value must be possible

Additive Generators

- Xn = arbitrary value, n < 55
- Xn = (X_{n-24} + X_{n-55}) mod 2^e, n \geq 55

Cryptographic Pseudo-Random Number Generators

- Given k bits of sequence, no algorithm can guess next bit better than 50% accuracy
- Infeasible to reconstruct the bits generated
- Block cipher - an encryption algo encrypts blocks of N bits

Java Classes

- Math.random() - random double in [0, 1)
- java.util.Random()
    - Random() generator with "random" seed
    - Random(seed) generator with given starting value (reproducible)
    - next(k) k-bit random int
    - nextInt(n) int in range [0, n)

Shuffling

- Shuffle is random permutations of sequence
- Possible algo
    - For every num in partition to swap
    - Swap element i-1 of L with element R.nextInt(i) of L
- Floyd Algo
    - For all nums to shuffle i = N-k, i < N
        - If some rand s 0 <= s <= i == some j
            - Add i to j+1
        - Else add to beginning

Week 14: Lecture 36 (4/22) Dynamic Programming

Dynamic Programming

- Start with list of non-negative integers

- Takes either leftmost number or rightmost
- to get largest sum

CS61A: memoization

- Memoize intermediate results
- Pass NxN memo array
- bestSum(int[] V, int left, int right, int total, int[][] memo) {
    - If (left > right) return 0
    - Else if (memo[left][right] == -1) {
    - Int L = total - bestSum(V, left + 1, right, total - V[left], memo)
    - Int R = total - bestSume(V, left, right - 1, total-V[right], memo)
    - Memo[left][right] = Math.max(L, R)
- From O(2^N) to O(N^2)

Longest Common Subsequence

- Problem: Find length of the longest string that is a subsequence of each of two other strings
- Int lls(String S0, int k0, String S1, int k1, int[][] memo){
    - If k0 == 0 || k1 == 0) return 0;
    - If (memo[k0][k1] == -1) {
        - If (S0[k0-1] == S1[k1-1]
            - Memo[k0][k1] = 1 + lls(S0, k0-1, S1, k1-1, memo);
        - Else
            - Memo[k0][k1] = math.max(lls(S0, k0-1, S1, k1, memo), lls(S0, k0, S1, k1-1, memo)

Enumeration Types

- Problem: Need a type to represent something that has a few, named, discrete values
- Public enum Piece {
    - BLACK_PIECE, BLACK_KING, WHITE_PIECE, WHITE_KING, EMPTY
- A piece is a new reference type where BLACK_PIECE are static final enumeration constants (enumerals)

Static Imports

- Import static checkers.Piece.*
- Will import all static definitions of checkers.Piece

Fancy Enum Types

- Enums are classes, can define constructors, methods
- Private final Side color
- BLACK_PIECE(BLACK, false, “b”)

Week 15: Lecture 37 (4/27) Threads, Garbage Collection

Threads

- Thread (“thread of control”) - each sequence
- Can run multiple threads, lock objects, wait, and interrupt
- Use java.lang Threads

Why

- Have some thread doing computation, thread for mouse clicks, update GUI

Java Mechanics
- Class Walker extends Thread {
    - Public void run() {
        - While (true) Walk();
- Thread clomp = new Walker2()
- clomp.start()

Avoiding Interference
- Can use void f() {
    - Synchronized (this) {

Communicating the Hard Way
- Faster party must wait for the slower
- Wait method makes thread wait(not using processor) until notified by notifyAll

Coroutines
- Coroutine: synchronous thread that hands off control to other coroutines so that one executes at a time

Use in GUIs
- Java runtime library waits for events and can designate an object as listener

Interrupts
- Interrupt is an event that disrupts the normal flow of control
- Does not receive the interrupt until it waits: wait, sleep, join
- Can throw InterruptedException

Remote Mailboxes
- Allow mailboxes in one program be received from or deposited into another

Scope and Lifetime
- Scope of declaration is portion of program it applies to
    - Static: independent of data
- Lifetime (extent) of storage is portion of program execution which it exists
- Static: entire duration of program
- Local or automatic: duration of call or block execution
- Dynamic: From time of allocation (new) to deallocation

Under the Hood: Allocation
- Java pointers are represented as integer addresses

Explicit Deallocating
- C/C++ require explicit deallocation
    - Lack of run-time information about what is array
    - Possibility of converting pointers to integers
- Java avoids by automatic collection
- Explicit freeing can be faster but error prone
    - Memory corruption
    - Memory leaks

Free Lists
- When storage is freed, added to a free list to be recycled

- Sequential fits
    - Link blocks in LIFO or FIFO stored by address, search for first fit
    - Segregated fits: separate free list for different chunk sizes
    - Buddy systems: segregated fit where some newly adjacent free blocks are easily detected and combined

Generational Collection

- Two or more spaces: one for newly created objects (new space) and one for “tenured” objects that have survived garbage collection (old space)
- Typical garbage collection collects only in new space, moves objects to old space