Midterm 2 Lecture Notes

Lecture 15-16

### Exceptions

Run-time error: Java “throws an exception” (Exception object). Prevent the error by “catching” the Exception

1. Coping with errors

try {

f = new FileInputStream(“~cs61b/pj2.solution”);

i = f.read();

}

catch (FileNotFoundException e1) {

System.out.println(e1); ///Exceptions have toString()

}

catch (IOException e2) {

f.close();

}

* Executes the code inside the try clause
* If try code executes normally, skip catch clauses
* If try code throws an exception, do not finish try code. Jump to first catch clause that matches exception; execute the catch body. Any code that throws an exception will throw a specific exception class type that the catch clauses look for

FileNotFoundException & IOException are subclasses of exception

2. Escaping a Sinking Ship—throw your own exception;

public class ParserException extends Exception {}

public ParseTree parseExpression() throws ParserException{

if (somethingWrong) {

throw new ParserException();

}

Need to declare what type of exception the class throws.

Exception vs return: Exception does not let you return anything. An exception can propagate down the stack through many stack frames. Will propagate until something catches it, or until the end of main in which case there is an error.

public ParseTree parse() throws ParserException, DumbCodeException {

p = parseExpressoin();

public void compile() {

ParseTree p;

try {

p = parse();

p.toByteCode();

} catch (ParserException e1){

} catch (DumbCodeException){}

Checked and Unchecked Throwables

throwable has two subclasses: exceptions and errors

Exceptions: IOException, RunTimeException( NullPointerException, ClassCastExceptions), etc.

Error: VirtualMachineError, OutOfMemoryError,AssertionError

Errors are things you should not try to catch. You need to be able to fix errors immediately, because they will most likely crash the program

Unchecked: errors and exceptions that you don’t have to declare, they will always be thrown when they come up. Almost any method can throw a NullPointer, ClassCast, etc.

Checked Method: a throwable that you must declare yourself.

“Finally” Keyword

f = new FIleInputStream(“filename”);

try {

statement X;

return 1;

} catch (IOException e) {

e.printStackTrace();

return 2;

} finally {

f.close();

}

If “try” statement begins, the “finally” clause will execute at the end, NO MATTER WHAT.

If statement X does not cause an exception. So try returns 1. But the return statement does not stop the finally clause; it executes the finally clause and then the entire method returns the value of 1.

If statement X throws an IOException, then the catch clause is executed. Then the finally clause is executed, and then 2 is returned.

If statement X causes some other exception, first the finally clause executes, then exception continues down the stack. The finally clause delays the exception but does not stop it from being thrown and propagating down the stack

Return in the finally clause? the entire return value of the method is replaced by the finally return

Exception thrown in “catch” clause: terminate the “catch” clause, execute “finally”, exception continues down stack.

Exception thrown in “finally” clause: replaces old exception, “finally” clause & method end.

Inside a catch clause or final clause, you can have another nested pair of try and catch statements.

Exception constructors

Convention: most throwables have 2 constructors.

class MyException extends Exception {

public MyException() { super();}

public MyException(String s) { super(s); }’

Generics

declare general classes (class that can store any type of object) that produce specialized objects.

SList for only strings, SList for only integers, but only one SList class

SList takes a type parameter.

class SListNode<T> {

T item;

SListNode<T> next;

SListNode(T i, SListNode<T> n) {

item = i;

next = n;

}

}

public class SList<T> {

SListNode<T> head;

public void insertFront(T item) {

head = new SListNode<T>(item, head);

}

}

Create an SList of Strings:

SList<String> l = new SList<String>();

l.insertFront(“Hello”);

Advantages: compiler ensure at compile time that nothing but strings enter your SList<String>. Makes it easier to debug than a class casat exception later if you try to use an item in a list that isn't the same as all the other items(say someone accidentally put an integer into a list only mean for strings without you knowing….hmm….)

Filed Shadowing

fields can be shadowed in subclasses. This is different from overriding. Choice of methods dictated by dynamic type. Choice of fields dictated by static type

class Super {

int x = 2;

int f() {

return 2;

}

}

class sub extends Super {

int x = 4; ///Shadows Super.x

int f(){ /// overrides Super.f()

return 4;

}

}

Sub sub new Sub()

super supe sub;

int i;

i = supe.x; //2

i = sub.x ///4

i = ((super) sub).x //2

i = ((sub) supe).x // 4

Lecture 17

### Game Tree Search

Alpha-Beta Pruning

public class Grid {

public Best chooseMove(boolean side) {

Best myBest = new Best(); // My best move

Best reply; // Opponent’s best reply

if ("this" Grid is full or has a win) {

return a Best with Grid’s score, no move;

}

if (side == COMPUTER) {

myBest.score = -1;

} else {

myBest.score = 1;

}

myBest.move = any legal move;

for (each legal move m) {

perform move m; // Modifies "this" Grid

reply = chooseMove(! side);

undo move m; // Restores "this" Grid

if ((side == COMPUTER &&

reply.score > myBest.score) ||

(side == HUMAN &&

reply.score < myBest.score)) {

myBest.move = m;

myBest.score = reply.score;

}

}

return myBest;

}

}

To turn this insight into an algorithm, we pass two additional parameters to the chooseMove() method: α and β. The pa- rameter α is a score that the computer knows with certainty it can achieve; for instance, if α = 0, then the computer knows it can force a draw, and is only interested in searching for moves guaran- teed to do better. Conversely, β is a guarantee that the opponent can achieve a score of β or lower. For any grid, we maintain values of α and β based on our current knowledge of the best moves dis- covered thus far. If β becomes equal to or less than α, then further investigation of the current grid is useless.

public Best chooseMove(boolean side,

int alpha, int beta) {

Best myBest = new Best(); // My best move

Best reply; // Opponent’s best reply

if ("this" Grid is full or has a win) {

return a Best with the Grid’s score, no move;

}

if (side == COMPUTER) {

myBest.score = alpha;

} else {

myBest.score = beta;

}

myBest.move = any legal move;

for (each legal move m) {

perform move m; // Modifies "this" Grid

reply = chooseMove(! side, alpha, beta);

undo move m; // Restores "this" Grid

if (side == COMPUTER &&

reply.score > myBest.score) {

myBest.move = m;

myBest.score = reply.score;

alpha = reply.score;

} else if (side == HUMAN &&

reply.score < myBest.score) {

myBest.move = m;

myBest.score = reply.score;

beta = reply.score;

}

if (alpha >= beta) { return myBest; }

}

return myBest;

}

Lectures 18-19: encapsulation and encapsulated lists, see homework and project

Lecture 20

### Asymptotic Analysis

Inventory

* 10,000ms to read inventory from disk
* 10ms to process each transaction
* n transactions takes (10,000+10n) ms
  + n plays a bigger part in the computation time when n is large

Big O Notation

puts an upper bound on computation time.

n is the size of the program’s input.

n can be word size, list length, bit size

Let T(n) be a function that models the running time in ms, or the memory allocation.

Let f(n) to be another function similar to T(n), but is preferably much simpler.

T(n) is O(f(n)) if and only if T(n) <= cf(n) whenever n is big and for any constant c

How big is n? c?

c and n can be any constants (as big as possible) as long as T(n) is still less than cf(n).

c cannot change as n changes. We can pick c and n arbitrarily.

EX:

T(n) = 10,000 + 10 n (actual running time)

consider f(n) = n

pick c = 20

T(n) and cf(n) intersect at n=1000. For n>= 1000, T(n)<=cf(n)

Therefore T(n) is in O(f(n))=O(n)

**Formal Definition of Big O Notation**: O(f(n)) is the **set** of all function T(n) that satisfy: there exists positive constants **c** and **N** such that for all values n>= N, T(n)<=cf(n)

In the above example, N=1000.

O(n) is a subset of O(n^2)

T(n)=1000000n is in O(n) with c = 1000000 and N = 0

O(2n) = O(n) unnecessary but not wrong

n^3 + n^2 + n is order O(n^3)

Proof: c = 3, N = 1

Big O notation only shows dominant polynomial term

Important Big O Sets

these sets are most often used in algorithms/data structure analysis. From smallest to largest (slowest to fastest growing).

| FUNCTION | NAME |
| --- | --- |
| O(1) | constant |
| O(logn) | logarithmic |
| O(log^2(n)) | log squared |
| O(n^1/2) | root n |
| O(n) | linear |
| O(nlogn) | nlogn |
| O(n^2) | quadratic |
| O(2^n) | exponential base 2 |
| O(e^n) | exponential |

Log to any power is still slower growing than root n, or any n to a constant power.

Algorithms running in O(nlogn) time or faster are considered efficient

n^7 time or more is useless

logn (base 2) is usually smaller than 50 in practice

Lecture 21 and 22

### Dictionaries

Two-letter words and defintions.

Word is the key that address the definition.

26\*26 = 676 possible words

To insert a definition into a dictionary:

function hasCode() maps each word (key) to integer 0…675

public class Word{

public static final int LETTERS = 26;

public static final int WORDS = LETTERS\*LETTERS;

private String word;

public int hashCode() {

return LETTERS\*(word.charAt(0) - ‘a’) + (word.charAt(1) - ‘a’);

}

}

public class WordDictionary {

private Definition[] defTable = new Definition[Word.WORDS];

public void insert(Word w, Definition) {

defTable[w.hashCode()] = Definition;

public Definition findWord(Word w)

return defTablep[w.hashCode()];

}

Hash Tables (most common implementation of dictionaries)

n: number of keys/words that we want to store in the dictionary~400k in English

Table of N buckets, where N is a slightly larger than n

Hash tables maps possible keys into N buckets by applying a compression function to each has code.

h(hashCode) = hasCode mod N

two different words can map to the same bucket— this is called a “collision”

Each bucket references a linked list of definitions, called a **chain**

how do you know which definition corresponds to which word?

each list node stores a word and its definition side by side

Hash Table operations (Goodrich and Tamasia)

public Entry insert(key, value) { //reference to a word and definition

compute keys hashcode

compress to determine bucket

insert entry into bucket’s list

}

Hash the key: compute key’s hashcode and sort into a bucket

public Entry find(key)

hash the key

search list for entry with given key

If found, return entry

public Entry remove(key)

find the key, then remove its

project 2 idea: key is gameboard, definition is score

will decrease computation time

2 entries with same key?

1. G&T: insert both— you have 2 words with the same key but difference definitions

find should return either definition, arbitrarily

remove should remove either arbitrarily

findAll can return all definitions of a key, not just one

2. inserting a key that already exists replaces the old definition with the new one. Only one entry with the given key exists.

load factor of a hash table is n/N

if the load factor of the hash table is low (less than 1)—and hash code and compression are good enough to avoid extraneous collisions and duplicate keys—then the chains are short, so each operation takes constant time.

If the load factor is large, ie n>>N, then the hash table takes O(n) time.

Hash codes and compression functions

key ————> hash code ————> [0,N-1]

ideally, map each key to a random bucket

Bad compression function:

suppose keys are ints

hashCode(i) = i;

compression function h(hashCode) = hashCode mod N

N = 10000

suppose keys are only divisible by 4 for some reason—we only use a quarter of the buckets

mod N where N is prime is a good compression function

better:

h(hashCode) = ((a\*hashCode + b) modp) modN

a,b, p : positive integers

p: large prime, p>>N

now number of buckets N does not need to be prime

**Ex:** Hash code for strings:

private static int hashCode(String key) {  
 int hashVal = 0;

for (int i = 0; i < key.length(); i++) {

hashVal = (127\*hashVal + key.charAt(i))%16908799;

///kind of like a base 127 number, but with mod for large #s

///16908799 large and prime

}

return hashVal;

}

multipier and modulus number must not have any common factors

Resizing Hash Tables

If load factor n/N gets too big, we lose constant time computation

enlarge has table when load factor > = about .75

allocate new array at least twice as long

walk through entries in old array, rehash them into new

we can also shrink a hash table when the load factor is very small in order to free memory, although this will speed up our computation time. Extending a hash table tends to be more important than shrinking one.

Therefore, the hash table should know how many buckets it has, how many entries it has at anyone time, and be able to calculate the load factor in order to know when it needs to expand.

Average running time over long run is still O(1) per operation.

**Transposition tables: speed game trees**

some grids are reachable through many different sequences of moves. Minimax algorithm means you need to evaluate the same game board many times, which can be time costly. We can create a hash table of previously stored grids.

Stacks

sample application: verifying matched parentheses

{[(){[]}]()}

algorithm: scan through string, char by char

left side parenthesis: push it onto stack

right sight parenthesis: pop its counterpart from stack, check that they match

If parentheses are not matched, mismatch or null returned or stack not empty at end of string

Queues

* “enqueue” item at back of queue
* “dequeue” item at front of queue
* examine “front” item

(a,b) ——> dequeue() ——> (b) ——> enqueue(c) ——> (b,c) ——front()——> b

sample application: printer queue

all methods run in constant time

implemented as a singly-linked list with a TAIL pointer, in order to constant time remove and add queue items to end

DEQUE (pronounced DECK)

Double-Ended- QUEue

you can insert and remove items at both ends by using a doubly linked list.

Lecture 23

### Asymptotic Analysis

Ω(f(n) is the set of all function f(n) such that there exist positive constants d and N:

for all n>= N, T(n)>= d\*f(n)

omega is the revers of big oh. If T is an element of O(f(n)) then f is an element of Ω(T(n))

2n is an element of Ω(n) because n is an element of O(2n)

n^2 is an element of Ω(n) because n is and element of O(n^2)

n^2 is an element of 3n^2 + nlogn because 3n^2+nlogn is an element of O(n^2)

Omega says your program is at least Ω(n) bad

Big theta precisely specifies functions asymptotic behavior.

ø(f(n)) is the set of al functions tat are in both of O(f(n)) and Ω(f(n)).

——> ø(f(n)) = O(f(n)) /\ Ω(f(n))

if f(n) is an element of ø(g(n)), then g(n) is in ø(f(n))

ex:

f(n) = n(1+sin(n))

f(n) is in O(n)

f(n) is in Ω(0)

f(n) is in ø(n(1+sin(n)))

—> f(n) has no simple big theta

Algorithm Analysis

Problem 1: given p points, find pair closest to each other

Algorithm 1: calculate distance between each pair, return minimum

there are p(p-1)/2 pairs

each pair takes constant time to examine. Running time is ø(p^2)

Problem 2: smoosh array

algorithm 2:

n is length of array

2 indices, i and j

i iterates up to n times

*ø(n)*

*Functions* of several variables

Problem3: matchmaking program for w women and m men

algorithm3: compare compatibility of each possible couple

compatibility computation is constant

Running time T(w,m) = ø(wm)

there exists constants c and d, and M and N such that

dwm <= T(w,m) <= cwm

for every w>= W and m>= M

T is not theta or omega order w^2 or m^2

These possibilities preluded by w>>>m or m >> w

You can’t asymptotically compare wm, w^2, m^2

Lecture 24-25

### Trees

Tree: a set of nodes, where any two nodes have exactly **one** path between them

Path: A sequence of one or more nodes, each consecutive pair connected by an edge

Rooted Tree: one distinguished node is a **root**. Every node c, except the root, has one **parent** p, the first on the path from c to the root.

If c is p’s child, then p is c’s parent for any two nodes

Root has no parent

a node can have any number of children but only one parent

Leaf: a node with no children

Internal node: non-leaf node

Sibling Nodes: have the same parent

Ancestor: the ancestors of d are all the nodes on the path from d to the root, including d itself and the root

Depth: the depth of the node is the length of the path from n to the root. Depth of the root is zero.

Height: length of path from node n to its lowest depth leaf. Height of a leaf is zero

Heigh of a tree is just the height of the root

Subtree rooted at n: tree formed by n and its descendants

Binary Tree: no node has more than 2 children, and every child is either a left child or a right child, even if it is the only child.

**Rooted Tree**

Each node stores 3 references:

item

parent

children—stored in a list

OR

sibling tree nodes are directly linked

class SibTreeNode {

Object item;

SibTreeNode parent;

SibTreeNode firstChild;

SibTreeNodenextSibling;

}

class SibTree {

SibTreeNode root;

int size;

}

**Tree Traversals**

Traversal: a manner of visiting each node in the tree once

Preorder traversal: visit each node before recursively visiting its children left to right. Starting at the root

class SibTreeNode {

public void preorder() {

this.visit();

if (firstChild != null) {

firstchild.preorder();

}

if (nextSibling != null) {

nextSibling.preorder(){

}

}

}

each node visited/preordered only once, so the pre order takes O(n) time

Lecture 26

### Binary search trees

every node has a parent, item, left child, and right child

Ordered dictionary: keys have a total order, like in a heap

quickly find entry with min or max key

Binary Search Tree invariant:

for any node x, every entry in x’s left child tree has to be less than x. Every entry

in x’s right child tree has to be greater than x

and in order traversal visits every node of the BST in order of value, from least to greatest.

Entry find(Object k) {

BinaryTreeNode node = root;

while (node != null) {

int comp = ((COmparable) k).

compareTo(node.entry.key());

if (comp < 0) {

node = node.left;

} else if (comp > 0) {

node = node.right;

} else {

return node.entry;

}

}

}

Entry remove(Object k);

find a node n with key k

return null is k is not in tree

if n has no children, detach it from its parents

if n has one child, move n’s child to to take n’s place

if n has 2 children, let x be node in n’s right subtree with smallest key

remove x. x has no left child so it is easily removed

replace n’s entry with x’s entry

Running Times

a perfectly balanced binary tree with height h has # of nodes n = 2^(h+1)-1

No node has depth > log\_2(n)

Running times of all ops are proportional to the depth of the last node encountered.

O(logn) time on balanced tree

all operations on binary search trees have theta(n) worst case time.

if you make a tree like 1-2-3-4-5-6-7 this will be O(n)

Lecture 27-28

### 2-3-4 Trees:

Every node has 2 3 or 4 children, except leaves, which are all at the bottom level.

Every node stores 1,2 , or 3 entries. the number of children is entries +1 or zero

1. Entry find(Object k)

like the normal binary tree search find

insert and remove are more complicated, because the restructure the tree

2. Entry insert(Object k, Object v) { //key and value

walks down tree in search of k

If it find k, it proceeds to k’s “left child” and continues

Whenever insert() encounters a 3 key node, middle key is placed in parent node. Parent has at most 2 keys and always has room for a third.

3. Entry remove(Object k);

Find key k

If it’s in a leaf, remove the key from the leaf

If its in internal node, replace it with entry with next higher key

remove() changes nodes as it walks down

Eliminates 1 key nodes except so keys can be removed from a leaf without leaving it empty

A 2-3-4 tree with height h has between s^h and 4^h leaves

The total number of entries n>= 2^(h+1) -1

h is in O(logn)

Time spent visiting node is constant time

All find insert and remove has O(logn) time worst case. This is versus a binary search tree that has O(n) time for all methods

GRAPHS

A graph G is a set V of vertices and a set E of edges that connect vertices.

Multiple copies of an edge are forbidden. Edges are unique.

A directed graph can have two edge connecting the same two vertices, because the edges can have opposite directions and therefore are not the same edge. (v,w) vs (w,v)

Self edges are also possible: (v,v)

Path: Sequence of vertices with each adjacent pair connected by an edge. If graph is directed, edges must be aligned with direction of path.

Length: of a path is the number of edges in the path

So a path between vertices in <4,5,6,3> has a length of 3

<2>: path of length 0

<2,2>: length 1

<2,2,2>: length 2

Strong connected: there is a path from every vertex to every other vertex. Usually applied to directed graphs. Otherwise a unidirectional graph is just called connected.

Degree of vertex: number of edges incident on vertex. Self edges count just once

For directed graphs, we talk about in degree and out degree

GRAPH REPRESENTATIONS

Adjacency matrix: ||V|| by ||V|| array of booleans (number of vertices by number of vertices). Maximum number of edges is |V|^2

Planer graphs have O(v) edges

graph is sparse if it has far fewer edges than maximum

Adjacency list:

Each vertex v has a list of edges out

1: 4

2:1

3: 2,6

4: 5

5: 2, 6

6: 3

Memory used is THETA(|V|+|E|)

If vertices are labeled by strings, you can use a hash table

GRAPH TRAVERSAL

A way of visiting every vertex once. Depth first search or Breath first search

DFS: searches a graph as deeply as possible as soon as possible. If graph is a tree, the DFS performs pre order traversals

BFS: visits all vertices whose distance from starting vertex is one, then two, and so on. If graph is tree, BFS performs level order traversal

Lecture 29

### public void BFS(Vertex u) {

u.visit(null);

u.visited = true;

q = new Queue(u);

q.enqueue(u);

while (q is not empty) {

v= q.dequeue();

for( each vertex w such that (v,w) is an edge in E) {

if (!w.visited) {

w.visit(v);

w.visited = true;

q.enqueue(w);

}

}

}

}

public class Vertex {

protected Vertex parent;

protected in depth;

protected boolean visited;

public void visit(Vertex origin) {

this.parent = origin;

if (origin == null) {

this.depth = 0;

} else {

this.depth - origin.depth + 1;

}

}

}

when edge (v,w) is traversed and we discover w is unvisited, we visit w, the depth ow w id the depth of v + 1, and v becomes parent of w

Albany

Kensington

Berkeley

Emeryville

Piedmont

Oakland

A: 0

K: 1

B: 1

E: 2

O: 2

P: 3

Shortest distant from Piedmont to Albany is P to O, O to B, B to A. Length of shortest path is just the depth of P, which is just 3. The shortest path can be found by following parent pointers

BFS runs in O(|V| + |E|) with adjacency list

For adjacency matrix, takes O(v^2)

Weighted Graphs

Each edge labeled with numerical number which is usually called a weight (but might be called a cost, slightly different meaning). Could be the capacity of the route(ie traffic, pipes, etc).

Adjacency matrix: array of ints or doubles.

Source are rows

columns are destination

For digraph, you need 2 adjacency matrices

Adjacency list: list of lists, where the list nodes has a weight field.

Problems

Shortest path: just like a google maps problem, for example, berkeley to LA

each edge is labeled with a rough approximation for the time, which is the weight

the shortest path is the path with the sum of the edge times is the shortest

BFS solves this is all edge weights are 1

BFS would not work on a map, because different edges take different time

CS 170: non uniform edge weights

Minimum spanning tree:

for example, making the least amount of wiring to outlets from electrical source, by chaining/ series. Each vertex is outlet or source of electricity. We use a graph

that shows all the ways we could possibly connect all the outlets to the source. Edges are labeled with the length of the wire, which is proportional to its cost ($). Connect all nodes with the shortest length of wire is the goal.

Kruskal’s Algorithm

G = (V, E) undirect graph

A spanning tree T = (v,F) of G is a graph with same vertices as G, and |v|-1 edges of that form a tree

If G is weighted, we want the minimum spanning tree, which is the tree with the least sum of edge weights. There could be more than one minimum, but they have the same total weight, so we use first one.

Create a new graph T with same vertices as G, but has no edges to being with.

We add edges one by one until we have a tree

We need to decided which ones to add and which ones not to add

Make a list of all edges in G, and sort by weight. Just reorder the adjacency list into one linked list with ordered weights. First make the list the sort from least to greatest.

Yes if adding it makes the graph a tree/forest still, no if it makes a loop. Iterate through the edges in sorted order from smalled to biggest. For each edge (u,w):

check: if u and w are not connected by a path in T, then add the edge because it is the least edge that can connect them. If they are connected, don’t add the edge to T.

T is always a tree (if G is connected). If G is not connected, then T is going to be a forest. So T will never have a loop.

Proof of G&T on page 649

Lecture 30-31- Midterm on Monday

### Sorting

Insertion Sort:

O(n^2) time

Insertion enforces sorted order invariant, only inserts in correct place

Start with empty list s and unsorted list I of n items.

for (each item x in I) {

insert x int S in sorted order

}

Linked list insertion: Theta(n) worst case time to find right position

Array insertion: theta(n) worst case time to shift items over

Run time proportional to the number of inversions, plus n

Inversion in an array is any pair of numbers out of order. 19 7 5 has 3.

If S is array, insertion sort is an in place sort

If S is balanced search tree, the running time is O(nlogn)

Selection Sort

runs in quadratic time (worst case and best case are the same). Invariant: sorted list S

Start with empty list S and unsorted list of I of n items.

for (i = 0; i<n; i++){

x = item in I with smallest key

remove x from I

Append x to end of S

}

Whether S is array or linked list, Theta(n^2) time, even in best case

Arrays still happen in place

Heapsort

Selection sort in which I is a heap

Start with S and I

toss all items in I onto heap h. Ignore the heap order property

h.bottomUpHeap();

for (i = 0; i<n;i++) {

x = h.removeMin()

Append x to end of S

}

Takes O(n) time to do bottomUpHeap()

Takes n for loops

removeMin() takes O(logn) time

Heap sort runs in O(nlogn)

Heapsort works in place:

maintain heap backward

Good for arrays, weird for linked lists

MergeSort

Merging 2 sorted lists into one sorted list takes linear time

Let Q1 and Q2 be 2 sorted queues. Let Q start as an empty queue

While (neither Q1 nor Q2 is empty) {

item1= q1.front();

item2 = q2.front();

move smaller of item1 and item 2 from p resent queue to end of Q

}

Concatenate the rest of nonempty queue to Q

Merge sort is a divide and conquer recursive algorithm

Start with unsorted list I of n items.

Break I into 2 halves I1 and I2. I1 has n/2 ceiling, I2 has n/2 floor items.

Sort I1 recursively, yielding S1.

Sort I2 recursively, yielding S2.

Merge S1 and S2 into S

Base case is a list of length 1. That base case is already sorted.

1+ceiling(log2(n)) levels. Every level takes linear work

O(n) time per level

O(nlogn) time total

Not possible to do merge sort in place

Quicksort

Recursive divide and conquer algorithm

Fatest comparison based sort for arrays

Theta(n^2) worst case time

almost always runs in O(nlogn)

Start with listI of n items

Choose pivot item v from I

Partition I into 2 unsorted lists I1 and I2.

I1: all keys smaller than v

I2: all keys great than v

Items with same key as v go in either list

Pivot v does not go into either list

Quicksort I1, yielding S1

Quicksort I2, yielding S2

concatenate S1,v and S2 to make sorted list S

Best way to choose the picot is to randomly select an item from I.

On average, you can expect a 1/4-3/4 split

Linked Lists:

Suppose we put all items with the same key as v into one of the lists, say I1. This will work, but it does not run fast.

Better: partition I into 3 lists. I1,I2, Iv. Iv contains pivot v and all items with the same key as pivot v.

Quick sort on Arrays:

In place

Input: array a; sort items a[low]…a[high]

choose pivot v; swap it with last item, a[high]

i = low -1

j = high

i and j sandwich item to be sorted

Invariants:

all items at or left of index i have a key<=pivot

all items at or right of index j have a key>= to pivot

advance i to key>= pivot

decrement j to key<= pivot

swap items and i and j

Repeat until i>= j

Swap pivot back to the middle with i