Algorithm analysis:

Abstract classes:

at least one abstract method, some design and some implementation

interfaces:

pure abstract classes

pure design

return fact3Aux(n, 1);

}

private static int fact3Aux (int n, int result) {

if (n == 1) {

return result;

}

else {

return fact3Aux(n – 1, n \* result);

}

}

Let f(n) and g(n) be functions mapping non-negative

integers to real numbers. We say that “f(n) is in O(g(n))” if

there exists a real constant c > 0 and integer constant n0 ≥

1 such that:

f(n) ≤ cg(n), for all n ≥ n0

So, f(n)=O(g(n)) means f(n) is a member of the set O(g(n)).

Let f(n) and g(n) be functions mapping nonnegative integers

to real numbers. We say that “f(n) is in Ω(n)” if there is a

constant c > 0 and integer constant n0 ≥ 1 such that:

f(n) ≥ cg(n) for all n > n0

So, f(n)=Ω(g(n)) means f(n) is a member of the set Ω(g(n)).

Big-Omega describes the least amount of a resource that an

algorithm needs for some class of input.

Big-Omega denotes a lower bound.

“f(n) is in Θ(g(n))” if there are constants c1 > 0

and c2 > 0 and an integer constant n0 ≥ 1 such that:

0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0

In other words, the function f(n) belongs to the set of

Θ(g(n)) if there exist positive constants c1 and c2 such

that f(n) can be “sandwiched” between c1g(n) and c2g(n)

for sufficiently large n.

for Recursion: T(n)=T(n-1)+c or T(n)=T(n/2)+c etc.

merge sort: split and split array into individual integers or objects and then merge them into bigger and bigger arrays in order, recursive and base case is having the split finally into an array of one value

T(n)=2T(n/2)+cn T(n/2)=2T(n/4)+cn T(n)=4T(n/4)+2cn =8T(n/8)+3cn . . . =nT(n/n)+cn(log2n) =n+cn(log2n) O(nlog2n)

quick sort: partition and partition smaller and smaller sections of an array until the whole thing has been partitioned into the right order, recursive and base case is array of size 1. pick pivot to be boundary between upper and lower half of array. then recursively do the same stuff by calling the partition to both arrays

SLL

adding works with point

add to tail: make the last one point to a new last one and assign tail

**public** **boolean** add (E e) {

SNode<E> temp = **new** SNode<E>(e);

**if**(head==**null**) {

add to head:

SNode<E> temp = new SNode<E>(o);

temp.next=head;

head=temp;

if head is null at first:

head=temp;

head.next=null;

head=temp;

tail=temp;

**return** **true**;

}

**else** **if**(head==tail) {

tail=temp;

head.next=temp;

**return** **true**;

}

**else** {

tail.next=temp;

tail=temp;

**return** **true**;

**Stacks:** list with only one end, last in last out

**push(element);** add element on top **pop();** remove top element **peek();** examine top element **isempty();** **size();...**can't loop through, no indices, last in last out, must pop through while not empty to go through

**Queue:** two-ended, first in first out

**add(value);** *enqueue* add element to back **remove();** *dequeue* remove front element **peek();** examine front element **size(); isempty();...** usually linked list implementation-->utilize front and back pointers

**Binary Tree:** collection of nodes with element(or object reference) and left and right child pointers, top is root

leaf: node with no children height(of node): largest path from root to any node

max nodes at level m: 2^(m-1) nodes (m counts from 0 like indices)

Binary Search Tree: each element in left subtree is less than root, each element in right subtree is greater than root

search always cuts in half if balanced O(log n), just compare >(go right) vs <(go left), to traverse and go left and right accordingly

removing a node: remove leaf node, just set left or right pointer null

remove a node with a left or right child only, replace the removed node with their child

node with two children: 1) find smallest element(get min) of right subtree, or largest (get max) of left subtree cause either would fit as a new root and have no children 2)save element of node found in 1 remove that replacement node 3) set node value to be removed as replacement node value

**next();** returns element in succession, in order

**Priority Queues and Heaps:** collection of ordered elements that provides fast access to "highest priority" element

**add();** returns in priority order O(log n), **peek();** returns "highest priority" value O(1)

**remove();** removes "highest priority" value O(log n) **isempty(); clear(); size();**

minheap: P<=x for every element x with its parent p maxheap: P<=x for every element x with its parent p

add minheap: add to leaf node and compare upward and swap with parent until parent is not greater

as array: root: a[0] for a[i]: parent = a[(i-1)/2] or children a[2i+1] a[2i+2] leaf nodes = (n=size) a[n/2] to a[n-1]

last non-leaf nodes: a[(n-1)/2] Heap Sort: sort array of n elements, turn into heap then call remove n times

**hash table:** use **hashcode(x);** function to give index spot to data.. like x%M (M is size of array/num buckets)

dealing with collisions: just add to index until open spot found or use each buckets as lists have multiple items in one bucket

**add(o); contains(o); remove(o);**

**Maps:** use keys that point to values, key value pairs, only one of each key, but different keys can have same values

maps.put(key, value)

**put(k, v);** if there is no such key, put method adds the key and its value, if the key exists already, just replace the old value

C: & = memory address of, local variables go out of scope when method call pops off stack so use memory allocation and pointers to keep things in scope after altering them

**printf("format string", variables);** ex. printf("ID = %d, GPA = %f", id, gpa); .. %d decimal %f float and double %s string % c character

No **booleans**  0-->false non-zero-->true ... **scanf**("%d", &age1)-->user enters number for memory location of age1

**Structs:** Struct Point { int x; int y;} --> idea of classes ... Struct Point p1; p1.x=0; p1.y=0; ... if you put typed def before the struct header, then you can give the struct a name after the end bracket so you can create instances of it with just the name not "struct Point newThing"

**Pointers:** int\* p1 or int \*p1 -->pointer to an int p1=&x -->the pointer p1 points to x .. dereference pointer \*p1=7-->x becomes 7 because new value in memory locaton ... dangling pointer points to nothing, don't assign values to these b/c no spot for value

int \*p; p=malloc(sizeof(int)); --> allocates enough memory to fit an int

if you use local variables in method call and return that variable after modification, value lost when method call popped off stack, so use pointers as parameter to keep hold on memory, and can either call method with pointer or memory locations(&a)

**Arrays:** int x[100]; aray of size 100, can use index values similar to java x[0], x[1], x[2] \*x=1 is the same as x[0]=1

pointer to array points to first element, the array is made in memory in order, all the values are next to each other in memory

int \*px = x -->means px points to beginning of array ... \*(x+7)=42; and \*(px+7)=42; is the same as x[7]=42; and px[7]=42;

incrementing a pointer like px, with just move it to next memory location, if px = &a[0]; and do px++; now px points to &a[1]

allocate memory for array, pa = (int \*)malloc(arraysize\*sizeof(int)); need to cast malloc whatever type the pointer is

**Strings**: well there are no strings in C, only character array: char string1[] = "sneezy"; works

all strings terminate with null symbol "/0" if you create array before giving value, must make array one longer than word for this symbol. string1 <,>,= string2 will compare things in memory not about the actual words

**int strlen(char\* str);** returns length not inluding null symbol **int strncpy(char \*strTo, char\* strFrom, int n);** copies n characters from strFrom to strTo **int strcat(char \*strTo, char\* strFrom);** copies string in strFrom to end of strTo

**int strcmp( char\* str1, char\* str2);** compares strings lexographicallys, is str1 is before str2 in alphabet, returns <0

**Free**, **free(ptr);** frees memory at location of that pointer, do not call free to dangling pointer b/c will crash

**void\* memset(void\* dst, int c, sizet\_n);** sets n nytes of dst to value c and returns dst

**void\* memcpy(void\* dst, void\* src, size\_t n)** copy blok of n bytes from src to dst

**(\*ps).id = 1911;** this takes a pointer to a struct, then modifies the struct's id(where id is a field in the struct)

**ps -> id = 1911;** does same thing and is easier

**int(\*foo)(int)** foo is a function pointer to any functiont that takes an int (ex. foo(2);)

read a file: FILE\* fptr;

if ((fptr = fopen(“students.txt", "r")) == NULL) {

fprintf(stderr,"Error opening file\n");

}

int numStudents;

fscanf(fptr, "%d\n", &numStudents);

// ... read more from the file

fclose(fptr);

thread safe: BlockingQueue, ConcurrentHashMap, ConcurrentLinkedQueue

**typedef void(\*Funcptr)(int); -->** Funcptr foo; foo = myprint;

**Parallelism and Concurrency (Java)**

Threads are objects

they have a run method which does stuff

they have a start method, which actually gets them to do the run method

**start() –** makes a thread eligible for execution in a thread by putting thread in a “ready” queue

**run() –** the method that’s run in a thread by start()

**isAlive() –** boolean, returns true if thread is alive

**join() –** this thread waits/blocks until a specific thread called with join is done

**yield() –** a hint to the thread scheduler that a thread is ready to release “control”

**sleep() –** sleep for some specified time

**setPriority(); –** for priority of which threads are running next and scheduling

yield and priority are hints to java, it won't always follow those

If t1.join() is called in the current thread t2, then t2

waits for t1 to finish and starts running.

things might access the same data concurrently so must be careful, some data structures are thread safe, some aren't

public void synchronized increment() { /\* atomic code \*/ } and synchronized(this) { /\* atomic code \*/ }

keyword **synchronized** is used to acquire a lock on the monitor of an object, careful->deadlock(locking objects and waitin for access)