Java Advantages

* Garbage Collection
* Errors
* Java API has every data structure you need

CS 2150 Notes 9/1

* #ifndef INTCELL\_H
* #define INTCELL\_H
  + for the header class (forward define all your methods)
* functions = methods not in a class 🡪 once in a class = methods
* No class visibility in C++
* If you don’t define specific visibility, default is private
* Look at IntCell.h code to see example of a header class
* Int getValue() const;
  + Method is not modifying object
* IntCell(int initialValue = 0);
  + Constructor can pass an integer or not, if you pass one, value is that value you specified
  + Now if you do regular constructor IntCell m; 🡪 initialValue is already = 0
* Int IntCell::getValue() const {}
  + IntCell specifies for what class you are writing this method for
* IntCell::IntCell(int initialValue):storedValue(initialValue){
  + Only works for constructors
  + List a field (storedValue) 🡪 parenthesis 🡪 value you want
    - Sets storedValue to initialValue
  + Also could do IntCell::IntCell(int initialValue) { storedValue = initialValue;}
    - But this way is very common so you will see it

9/3

* #define PI 3.1415
  + define a constant with #define
  + can also just say #define FOO
    - no value, just defined
  + #ifdef FOO //this is true
  + #ifndef FOO //this is false
  + #endif
  + #endif
    - every ifdef has to have an endif
  + area = PI \* r \* r;
    - uses the constant PI
* Pointer
  + Memory address of another object
  + Can be a primitive type of a class type
  + Int \*x
    - Pointer to int
    - 32-bit spot in memory that is ready to hold a memory address
  + \*x = 2;
    - evaluates object to which the pointer points (pointee)
  + x = &y
  + “address of”
  + int x = 1;
  + int \*x\_pointer = 1;
  + x\_pointer = &x;
  + cout << x\_pointer;
    - prints out memory address of x
  + cout << \*x\_pointer;
    - prints out value of what’s at address of x\_pointer aka 1
  + \*x\_pointer = 2;
    - changes x to 2;
    - changing a value of the pointee through a pointer

9/5

* int \*x;
* x = new int;
  + no parentheses
* \*x = 42;
* gotta set up the pointee
* swap(int \*x, int \*y)
  + int temp = \*x;
  + \*x = \*y;
  + \*y = temp;
* have to deallocate memory that we are no longer using
  + keeps using it until program ends
  + good practice because imagine if you were writing a web browser, which you keep open for hours, computer would become unusuable because the memeory is all used
* \*x = new Rational(1); 🡪 not sure if this works…?
* Delete rPointer\*;
* Delete [] pointer;
* Anything allocated with new MUST be deallocated with delete
* Two ways to declare an object:
  + Way 1: Rational r;
  + R.num = 4;
  + Way 2: Rational \*r = new Rational();
  + (\*r).num = 4; == r->num = 4;
* char\* x, y;
  + everything on the line is declared as a character pointer
* char \*x, y;
  + everything on the line is a character, but x is a char pointer
  + associativity of \*

9/8

* Friend
  + Sometimes other classes need access to private data members of another class
  + Friend class List;
    - Now the List can access all of ListNode’s “private” fields
    - Now when you create a List, List can see everything in ListNode (List can set ListNode’s next, previous, etc)
    - However, ListNode cannot see everything in List
  + References
    - Like a pointer, holds an address
      * However, its address cannot change
      * Must be initialized upon declaration
      * Has implicit dereferencing

9/10

* New
  + Allocates a new memory location for use
* Delete
  + Reclaims memory allocated with new
* Big Three
  + Destructor
  + Copy constructor
    - IntCell copy = original;
    - IntCell copy(original);
    - Both lines above do the same thing
    - Creates a new object, initialized to a copy of the same type of object
  + Operator=()
    - IntCell original;
    - IntCell copy;
    - Copy = original //operator= called
    - After both objects have already been constructed
* Vector
  + Kinda like ArrayList
  + Problems with arrays
    - Can’t be copied with =
    - Does not have a notion of capacity (can’t do .length)
    - No index validity checking
      * If you try to put something in index 12 of a size 10 array, it won’t tell you that you can’t do it
  + Iterator
    - Begin()
    - End()
    - Itr++ //advances itr
    - \*itr //returns reference to object at itr location
    - itr1==itr2 //true if at same location
    - it1!=itr2 //true if different location
  + Templates

9/12

* If find(k) is an important method to you, use an array based implementation vs. linked list
  + Because linked list you have to go through each one with an iterator, while with array you can just use the indices
* Stacks
  + LIFO
  + Can only insert, delete and find at “top” of list or the head
  + Push: insert to top
  + Pop: delete from top
  + Top: examine top of the stack
  + Applications
    - Undo operation in programs
    - Operator precedence

9/15

* Queues
  + First in first out
  + Insert in the back (enqueue), remove from the front (dequeue)
  + Applications:
    - Print queues
    - Lines in general
    - File serving
    - Call queues
* Number Representation
  + Computers think in base 2

9/17

* 42 base 10 🡪 radix 5
  + 42 divided by 5 = 8 remainder 2
  + 8 divided 5 = 1 remainder 3
  + 1 divided 5 = 0 remainder 1
    - 132 base 5 is the number
* keep going until quotient is 0, then remainders in reverse order is the new number
  + only for base 10
* if a number begins with 0 it is an octal aka base 8
* if a number begins with 0x it is interpreted as hexadecimal
* big endian: most significant first
* little endian: most significant digit = last

9/22

* Mantissa:
  + ¼ + 1/16 + 1/256
  + 0101 0001
    - first 0 is ½ (it’s not represented by ½ so it’s 0)
    - second 0 is ¼ (it is represented by ¼ so it’s 1)
    - third 0 is 1/8 (not represented so it’s 0)
    - fourth is 1/16 (it is represented so it’s 1)
    - etc. etc.
* use the slide doing the last example by hand, part 1 for the prelab

9/24

* arrays
  + cannot be copied with =, must be done element by element
  + int\* someInts = new int[3]; 🡪 another way to declare an array
    - must use delete to delete it
  + int someInts[3];
    - compiler is allocates the memory vs. new where you allocate it
  + &someInts[i] = {addr of someInts} + (sizeof(int) \* i)
  + someInts == someOtherArray is allowed but only returns true if they are pointing to the same memory location
  + multi dimensional arrays are stored in row major order
* Big O
  + Functions that grow *no faster* than g
  + X axis is number of inputs (positive integer)
  + Y axis is run-time
  + Upper bound
* Big Theta
  + Functions that grow *at the same rate* as g
  + Will run in this time period, not upper or lower bound
* Big Omega
  + Functions that grow *at least as fast* as g
  + Lower bound

9/26

* Big o, theta, and omega are all worst case
* Going to use big theta
* Show that f(n) <=c\*g(n) for all values n >= n0
  + Pick a c 🡪 n <= 2n^2 🡪 for all n > 2
  + Pick an n0
  + 1 <= 2n for all n > 2
  + Because n is positive, this inequality holds true
* Is 10n <= o(n)
  + C = 11, nsubzero = 2 works so yes
  + Remove constants
* Is n^2 <= o(n)
  + No, no values for c and nsubzero n

Review Session

* #ifndef = if not defined
* c++ compiler doesn’t look ahead like java compiler does when defining methods and stuff
* .h is prototype
* separating interface and implementation
  + reason for using prototypes
  + many classes might have to use certain classes (ie. Listiterator.h)
    - another reason for .h
* copy constructor vs. operator =
  + list l;
  + l = m;
    - two separate statements
    - declare the list first, then assigning a value to another value
    - uses the operator =
  + list q = r;
    - declaration and assignment in same statement
    - uses copy constructor
    - assigning is being copied same time as being constructed 🡪 copy constructor
* pointers vs. references
  + current (list class) needs to be pointer cuz when you call next, you need to go to the next one
  + references: reference is a pointer
    - but reference’s value can never be changed (cannot point to anything else)
    - must be initialized upon declaration (makes it so you have to point to something valid)
    - implicit dereferencing
  + references useful as paramaters because the thing they are referencing are initialized outside of function vs. if you do it in the function you could just use the original item
* passing by value vs. passing by reference
  + when pass by value, it makes a copy of the thing and gives it to function
    - uses copy constructor
  + but if you have a huge object, spends a lot of time copying all the stuff into the subroutine
  + pass by reference:
    - formal paramaters are addresses of actual paramaters
      * formal parameter = objects in the function
      * actual parameter = l.foo(4); in main
    - now formal parameter is modifying actual parameter
* when would you pass a “blah” by reference
  + want a modification to formal parameter to change actual parameter
* unions
  + writes both the float and int \*x to the same memory addresses
  + bar.f = 42.125
  + bar.x 🡪 reads it as an integer and cout knows to print it as a hex
* prove n^2 is big omega (n)
  + f(n) = n^2 g(n) = n
  + prove that f(n) element of omega (g(n))
  + have to show that f(n) >= c\*g(n) for all n > nsubzero
  + n^2 >= c\*n for all n > nsubzero
  + n >= c
  + let c = 1, nsubzero = 1
  + n >= for all n > 1
  + it is true that n >= 1 for all n > 1
* run time of vector insert
* linked list run time
* memory hole
  + int \*p = new int;
  + p = null;
  + p knew where that new int was, but now we don’t know where it is cuz nothing is pointing to it
  + have to deallocate that shit with delete command
* templates
  + if you want a linked list for a lot of stuff you can write a template and you straight
  + doesn’t have the types of the things it holds in paramaters and returns that’s why it’s not a class it’s only a template
  + purpose: type-independent code
    - vector that can work with any type
  + cause a lot of errors because hard to see where error is when compiling
* when you write a class, it comes with:
  + constructor (no paramaters tho)
    - empty body
  + operator =
  + destructor which does nothing
    - empty body
* destructor
  + if i just use delete thing for list, it only deletes the head tail and count thing for list not the listnodes that have been allocated that’s why you need a destructor
* mantissa
  + difference between precision and range?
* Iterator allows you to go through collection of things without worrying how to get to next one
  + For example, linked list uses next(), array you have to use for loop, tree set figures out how to move up and down trees
  + Theres an iterator in stl
* Abstract data types
  + Way to talk about a class without having specific to a given language
* Single precision: 32 bit (7 digits of accuracy)
* Double precision: 64 bit floating point (about 15 digits of accuracy)
  + Good for almost everything you need
* Quad precision (38 digits)
  + Usually don’t need that
* Implement a stack with a linked list
  + Insert is always constant, just create a node and push
  + Pop is always constant
  + Still slow
* Implement a stack with an array
  + Vector…if it’s full you have to make a new one, can be expensive
  + If you make one of a million things then remove 999,999 you are taking up a lot of space

9/29

* Little oh
  + Strictly less than f(n)

10/1

* Binary trees
  + No sorting, no order (no idea where data is, can be in any position)
    - If 1 is the root, and you want to find 4, you don’t know where it is
  + You can have very unbalanced trees
    - if you have data that’s already sorted
* Binary Search Tree
  + Binary tree with ordering property
  + For any given node, if value is less, it’s in the left subtree
  + If value is greater, it’s in the right subtree
  + Assume no duplicates
  + BST: Find
    - Compare value to be found to key of root
      * If equal 🡪 done
      * If not, recurse depending on which half of tree value should be
      * If you hit a null, value is not in the tree
  + BST: Insert
    - If(current == null) 🡪 insert there
    - If x < current
      * Recurse Left
    - If x > current
      * Recurse Right
    - If not
      * //duplicate don’t do anything
    - pass by reference because you want to modify the formal parameter (current) to effect the actual paramater
  + BST: remove
    - Find node to be removed
    - Fix tree so binary search tree still holds true
    - Three cases:
      * Node has no children
      * Node has one child
      * Node has two children

10/3

* Perfect binary tree
  + All leaves have same depth
  + And all nodes have zero or two children, but not one child
  + Number of leaves: 2^h
  + Number o nodes = 2^(h + 1) – 1
* For lab
  + Make stack of tree node pointers

10/6

* Templates have hard time because compiler generates code so it might not be a syntax error but it could be a design error like forgetting the operator=, so that’s why it has a hard time finding the problem
* AVL tree uses balance with balance factor
  + Right subtree – left subtree
  + Height and left and right subtree can be off by at most 1
  + Height: number of edges on longest downward path to leaf

10/15

* Separate chaining
  + All keys that map to the same hash value are kept in a “bucket”
    - Bucket is another data structure, typically a linked list
  + Worst case:
    - Everything hashes to exact same spot
      * Linked list with everything out there, linear runtime
  + Hash(k) = k mod 10
  + Find() is linear runtime
  + Never use vectors for buckets
    - Creates huge arrays for each cell, uses too much memory
  + Can stick as many elements as you want, but you will have long chains
* Load factor
  + Ratio of the number of elements divided by the table size
  + Want it to be less than one
  + As load factor lowers, hash table gets bigger
    - Fewer collisions but uses more memory
* Small number of elements 🡪 linked list is less faster than red-black tree
  + Expected number of inputs in a bucket for separate chaining is always small, therefore, always use linked list, therefore, find() is always linear (not logarithmic like red-black tree)
* Hopeful case
  + Good hash function: Even distribution, size of table scales linearly with number of elements, deterministic, fast
* Linear probing
  + Use that hash function, if it’s filled, try hash(k) + 1, if that’s full, try hash(k) + 2, etc. until you find a spot that’s empty
  + To find, do the same thing, probe successive spots until you find it
  + Can only stick n elements if n is size of hash table
  + Problems
    - Large blocks of occupied cells
    - As table fills, increased number of
* Quadratic probing
  + Same as linear probing
    - But instead of adding 1, 2, 3 🡪 add 1^2, 2^2, 3^2
  + Gets out of cluster faster, but has other problems still as linear probing like holes, etc.
  + Max number of elements you can insert is n, if load factor is high, it’s inefficient
* Double hashing
  + Add 1, 2, 3 multiplied by a second hash

11/5

* Paramaters are positive offset, local variables with negative offset from base pointer which is middle of activation stack
* Pushing a parameter decrements stack pointer by 4 then pushes the thing on the stack
* Activation record:
  + Caller saved record: eax, ecx, something else
  + Pushed paramaters
  + Local variables
  + Return spot when you say ret where it goes to
  + Two other things i don’t remember
* Leave opcode does a lot of clean up
  + Pops off variables and stuff
* Mov eax, DWORD PTR [ebp+8] ;load a 4 byte int into eax
* Movzx eax, BYTE PTR [eax] ;load the one byte character from eax into eax so that its just one byte
* DWORD PTR [esp]

Test 2 Review

* If you multiply a register, you’re aligning it for different words
  + Addressing scheme of
  + Mov eax, [4\*esi+edi+8]
    - Takes value out of that spot in memory
    - At most two registers, 2 or 4 or 8 \* esi
      * Can only multiply by those, nothing else
      * For quick array traversing
    - Array subscripting
    - [4\*esi + edx]
      * edx is base of array, increment esi by one in a loop
      * that helps you get every element in array
      * if it was 2 rather than 4, it’s an array of shorts or an array of characters if it’s 1
      * 8 if it’s doubles or longs
      * set up like this so you can do arrays quickly
        + “esi”th element of an array
  + You can access memory at most once in a single instruction in x86
* Accessing memory twice is invalid
  + Mov [eax], [var]
    - It’s trying to pull something out of memory, then move it into the spot in memory referenced by eax register
    - CPU doesn’t have enough time to perform that at internal clock
* Lea is load effective address
  + Lea eax, [var]
    - Loads effective address of var into eax
* Thrashing
  + Double hashing: hash all elements first, if you have a collision, use secondary hash function
    - Put it in that spot, but what if the secondary hash function keeps probing the same spots that are filled, but there are still spots open 🡪 causes infinite loop
      * Why you need a prime number table size
        + Want to make sure that secondary hash value never has values that are a factor of table size 🡪 make table size prime
* Quadratic Probing
  + If table size is prime, and spots in table available, guaranteed to find a spot
* Load factors
  + Trade off between memory use and running time
  + Separate chaining
    - You can have a load factor as big as you want
    - You’re still going to get inefficient running times
  + Usually keep load factor < 1
  + Linear probing, quadratic probing, double hashing
    - Load factor of 1 would be very inefficient 🡪 same thing as a vector
    - Lower load factor you have, more space you need for hashtable, fewer collisions
    - Higher load factor you have, less space, but more collisions
  + Also the probability that you have a collision aka .5 load factor = 50% chance you get a collision
* Bash and shell scripting are fair game for exam
* When passing command line arguments, why does argv argument have a pointer to an array rather than just an array
  + You could replace it with \*argv []
  + ./a.out foo bar baz
    - if that’s command line you type in 🡪 argc = 4
    - argv is an array where each item in array is a pointer to a character
    - not really super important tho
* Given that f(n) = 3n^2 + 7n – 3, prove that f(n) is an element of Theta (n^2)
  + Question 10 on fall exam 2012
  + Show that f(n) is big O (g(n))
  + Show that f(n) is big omega (g(n))
  + f(n) <= c \* g(n) for all n > n0
  + 3n^2 + 7n - 3 <= c \* n^2 for all n > n0
  + Pick any c and any n0 that would make that work, try picking c = 5
  + Show that 3n^2 + 7n – 3 <= 5n^2
  + Try picking n0 as 5
    - N can’t be 0 because there is nothing to compute, so minimum is 1
  + 7n – 3 <= 2n^2
  + 32 <= 50 🡪 check mark bitch
  + Big omega is same thing except that instead of <=, change it to >=
* How to do big theta for a hashtable
  + Big theta is not great for some things
  + Ie. Red-black trees and avl trees are both big theta (log n)
  + Same thing with hashtable, always look at worst case
    - Worst case with hashtable is linear, you have a linked list in separate chaining or like a vector in probing
* Which is more likely to have linear run time, binary search trees or hashtable
  + For a hashtable to be linear, everything has to hash to the exact same spot, not likely to occur
  + But for a binary search tree, if data is already sorted, its linear aka much more likely to be linear
* Difference between left and right subtree can differ by a factor of 2 at most
  + In red-black tree
  + Any simple path from a node to any descendant leaf contains the same number of black nodes
  + Both children of every node are black
* Avl trees can only have difference between left and right subtree by factor of 1
* Red black trees uses black layers that help enforce balance which help any one tree from getting too far down
* Don’t worry about insert cases for red-black tree, 5 of them
* Red-black is faster than an avl tree because it doesn’t have to spend as much time balancing the tree
  + Avl does a lot more tree rotations, they are slow
* Situations when avl tree is used over red-black tree
  + If you had to implement balanced binary tree yourself because it’s a new programming language
  + Not usually tho
* When you’re looking at avl tree rotation, don’t think intituitively, do the insert by normal bst insert, compute all balance factors, then look at lowest node that is unbalanced and do that
* Splay trees
* Ammortized time
  + In tree set after splay trees
* Invoking methods
  + Private vs. public visibility is not worried about in assembly, it’s checked at compile time by c++
  + If you invoke a method and pass 4 parameters, there is actually 5 parameters, where the first one is that object
* When you might not push ebp on the stack
  + If you are doing a max function and you don’t need to create any local variables, or an activation record
  + Esp + 4 is first parameter, esp + 8 is second parameter, that’s how you know where they are if you don’t want to upsh ebp
* For most things, a string is linear
  + You can find a spot in a string in constant timem like an array
    - Substring is length of string, run time so it’s constant
    - Indexing is also constant
    - Iteration is linear, still have to print n elements
  + But for ropes, because it’s using binary tree, worst case is that you have just characters and they are in order so it’s linear
  + This is from exam 2010 spring: ropes
* Optimization flags: only see –02
  + –02 🡪 try to make code run faster
    - don’t deal with local variables because it just uses registers instead
    - there’s more i didn’t catch them
    - loop on rolling
    - get rid of unused code, probably, would have to give an explanation tho
* Why are parameters passed in reverse order
  + Because: variable number of arguments
    - Subroutine that has one argument and then some number of arguments ie. Average(3, 12.2, 22.3, 4.5)
    - Or average(5, 3.3, 2.2, 1.1, 5.5, 3.3)
    - Need to know where first parameter is
    - If we did it by first order
      * We would push that first number, then the rest of the parameters, and now we don’t know where that first number is
    - If we do it in reverse order
      * First parameter is always at ebp + 8, and now we can do variable number of arguments
* Exam 2 spring 2014
  + Number 6
    - Compute the balance factors
      * Right – left
    - Insert six, it goes to left of 7
    - Update balance factors starting from that node to root
    - See that 4 (root node) has balance factor of 2
    - Have to rotate that 4
    - Look that direction was right, left
      * Double rotation
    - Do double rotation on 5-13-9
      * Rotate bottom first so that its 5-9-13 all right subtree
      * Then rotate again so you get 9 as root, 5 as left, 13 as right
    - Attach trees to that described above
    - Where tree A is just 3-1, B is 1-6, C is just 11, D is 17-15
    - Then attach everything back a, b, c, d
* Why do C and C++ use different naming conventions when doing subroutines
  + In C, can only have one subroutine named foo
  + C++ can have a lot of subroutines called foo
* Ways to prevent buffer overflow attacks
  + Character buffer of size 12 can hold string of size 12
  + Buffer overflow: where you have a spot in memory to write to (buffer), and you don’t have anything that checks how much data is gonna be read in
  + So if it’s bigger than 12, you get an overflow, it overwrites ebp
  + As it starts destructing register, it doesn’t have the right ebp, and if you do 20 bytes, you screw up return address
  + Prevented by between buffer and ebp, it puts a stack canary, put a specific 4 byte value on the stack, if you do a buffer overflow, you overflow that stack canary, and when it’s destructing register, it checks to make sure that the stack canary is still there
  + Can’t override the stack canary,
  + Also modifying where stack starts rather than end of memory
* Stack grows down that starts at end of memory
* Maximum value held in a double
  + 1111 1110 is max value in float
    - 0 is sign bit
    - double has 11 bit for exponent rather than 8
    - exponent for this one is 254-127
  + 2\*2^126 = 2^128
* Big theta = in between big O and big omega
  + Aka big theta is “equal to”
  + Little O is strictly less than, little Omega is strictly greater than
  + You can’t be strictly less than and strictly greater than 🡪 no little theta
* What changes would you have to make in ibcm to allow it to implement any algorithm
  + Limitation of insufficient computer resources (always exists)
    - Have to give it infinite memory
  + IBCM can do any algorithm
* Useful flags
  + -02: optimized executable
  + –g: debugging
  + –Wall
    - display warnings
  + –S: generate assembly output
  + –mlvm --x86--asm=intel
    - set assembly output to intel
  + –c
    - compile but not link the .cpp file
  + –m32
    - run as runs as 32 bit program
* Makefiles
  + Target: name for something that make should do
    - Can be a program
    - A debug version of a program
    - An optimized version of the program
    - Always has a colon after it
  + Dependency
    - File that is either directly or indirectly included
    - When you compile one file, you want to have the other effected or dependent files compiled as well
  + Suffix rule
    - .o .cpp
      * means create a .o file from every .cpp file that is named the same

11/12

* Binary heap
  + Binary tree not BST with different structure and ordering
  + Used to implement a
* Huffman Lab
  + Int array from 0 to 255

12/5

* New keyword is slow compared to just an array
* Cache
  + First time something is pulled out of main memory, it’s put in the cache
    - Every time it’s needed next, gets it from cache
    - Holds a limited subset of memory
      * Smaller than main memory so you can’t hold everything, but most recent/frequently stuff from memory
    - This works because getting from main memory is slow
  + Content at each level is a subset of the level below
* Trying to get something out of memory
  + Main memory is in pages, so if you just want like 4 bytes thing, main memory only gives you pages so it gives you the whole page
    - One of the reasons why it’s slow because a page is 1 Kb
* Going through a two dimensional array
  + Have to go through with nested for loops, make sure inner loop is second thing ie. Array1[i][j] 🡪 j needs to be inner most loop
  + Make sure i is outer most loop
  + This is so that it doesn’t go super slow because you want to go in row major order like how arrays are represented in emmory