David Cole

Review

Chapter 1

1. Objects are instantiated parts of a class, they come below them
2. Inheritance
   1. Allows you to create a new class that augments an existing class
   2. Can keep your code separated and organized, and allows reusing code.
   3. New class is called sub class
   4. Java only has single inheritance
      1. Only can move up the inheritance tree
      2. All classes can be traced up to the Object class
3. Direct vs Indirect addressing
   1. Direct – the variable points directs to a location in memory where the information is stored
   2. Indirect – the variable points to a location in memory that points to where the information is stored.
4. Order of growth
   1. O(N) – operations depend on the size of the input
   2. O(I) – size of the input is irrelevant
   3. O(Log2N) – logarithmic growth uses binary search
   4. O(N2) – two nested for loops

Chapter 2

1. Abstraction
   1. Hiding details that are not necessary to use an item
      1. An example would be a helper method that helps a user use a program
2. Java Interface
   1. Formal definition of ADT
   2. Allows declaration of variables and methods
   3. All variables must be constant and all methods abstract
3. Stack
   1. Fast access to element on top
   2. LIFO behavior
   3. Methods
      1. Push – adds element to the stack
      2. Pop – takes the top element off of the stack
      3. Top (peek) – allows you to get info of top element
   4. If we wanted to store multiple types of data could make a stack of type object store the diff data in the object then store the objects in the stack.
   5. Post fix evaluator
      1. Postfix = 4 5 7 2 + - +
         1. Infix – 4 \*(5-(7+2))
      2. Postfix = 3 4 + 2 \* 7 /
         1. ((3 + 4) \* 2 ) / 7
      3. Postfix = 5 7 + 6 2 – x
         1. (5+ 7) \* (6 – 2)
   6. Linked Lists
      1. Series of nodes that contain data and a link to the next node
      2. The nodes are stored randomly throughout memory
      3. OOG Linked list
         1. Accessing 1st element – O(I)
         2. Last element – O(N)
         3. Middle = O(N)
         4. Inserting at front – O(I)
         5. Inserting middle – O(N)
         6. End – O(N)
      4. Linked lists are more memory efficient than arrays
      5. Arrays are more time efficient
      6. Bad for random access – no index?
   7. LinkedStacks
      1. Stackinterface methods
         1. Push – O(I)
         2. Pop – O(I) – both removes and returns
         3. Top – O(I) – technically don’t need because we could pop store it then push it back on
         4. isEmpty – O(I)
         5. isFull – O(I)
      2. best implementation of linked list is when it requires being resized a lot

Chapter 3 Recursion

1. important part of recursive definition is checking for invalid inputs in the beginning
2. 3 pieces
   1. Base case
   2. Check for base case
   3. Recursive case
3. Recursion uses more memory than loops due to saving methods on the call stack
4. 3 parts of check
   1. Base case check – to see if base case is operating correctly
   2. Base case operation – does the function approach the base case
   3. Recursive operation – does the function perform the recursion correctly
5. Binary search
   1. Good for recursion because can cut amount to search in half with each call
   2. Checking the middle of the range with every step, requires array being sorted
      1. Found
      2. Less
      3. Greater
   3. O(Log2N)
6. Recursion with linked lists
   1. Linked lists possess a recursive structure
      1. LLNodes being self referential
   2. To iterate through simply have if (iterator != null)
      1. SOP(iterator.getInfo());
      2. Recprintlist(iterator.getLink());
   3. Modifying linked list recursively
      1. 2 methods
         1. Void method that adds to the list and takes list and item as params
         2. Return a pointer to the modified version of the list
            1. Simplifies recursive code
            2. More flexible if large changes are being made
            3. Uses call stack to maintain needed pointer state
7. Static vs Dynamic
   1. Static – memory is allocated when the program is compiled and no more memory can be added – doesn’t support recursion
   2. Dynamic – memory is allocated as the program is being run – supports recursion
      1. Stored on the call stack – takes more memory
         1. Stack that ensures return statements are handled in correct order
8. Avoid recursion
   1. When can easily be done in a loop
   2. Can be done explicitly with a stack
      1. Instead of passing the method another parameter store that parameter on a stack then at the end pop and print the stack.
   3. Sometimes unnecessarily repeat computation
   4. Tail recursion
      1. Uses a loop to track what the call stack usually would have to do to save on memory
         1. Requires O(I) if done correctly
         2. Would only have one return statement

Chapter 4 Queues

1. FIFO data structure – first in first out
2. Methods
   1. Vital
      1. Enqueue – adds to the back of the queue
      2. Dequeue – both removes the item that’s been there the longest and returns it
3. Improving dequeue
   1. Track both the front and the rear – tail node
4. Circular structure – array based implementation
   1. Making it so when the queue is being dequeued that the tail loops around to the former front of the queue so that the queue will not run out of space
   2. Reduced OOG significantly
5. Unbounded queue
   1. Uses an enlarge method
   2. Created a new array then copies the elements over
   3. Has OOG O(N)
6. Linked Queues
   1. Two pointers
      1. Head
      2. Tail
   2. Enqueue
      1. Adds a node at the tail
      2. Must change tail to point to the new node
      3. O(I)
   3. Dequeue
      1. Head will now point to the element that the first node used to point to
      2. Change pointer of first element to point to null
   4. Could have only one pointer
      1. Have the tail also point back to the head
   5. Array based implementations are more space efficient when dealing with a large queue relative to the max size (break even around 50%)
   6. Small or variable queue size a linked list is more efficient
7. More variations
   1. Don’t need over/underflow exceptions
   2. Could just have isFull/ isEmpty return Booleans then use them in other methods
   3. Glass Queue
      1. Allows to peek at the top of the queue
      2. Used for when have multiple queues
   4. Deque
      1. Either queue or stack
      2. Enqueue and dequeue have 2 versions
         1. Front and back
   5. Doubly linked list
      1. Requires nodes storing 2 links (front and back)
         1. Requires more memory
8. Waiting time
   1. Subtract the finish time of the previous customer by the arrival time of the current one to find wait time

Chapter 5 Collections

1. Storing data in an unordered manner
2. Access is based on content
3. Critical methods
   1. Add() – Boolean indicator of success
   2. Get()
   3. Remove() - Boolean indicator of success
      1. Takes the last element before null elements
      2. Places that element in the spot of the element you just removed
      3. Done to ensure that there are no null elements (not allowed)
   4. Contains()
4. Helpers
   1. isFull()
   2. isEmpty()
   3. size()
   4. find()
      1. most important helper method
      2. used in remove, get, contains
      3. find an element and returns the pointer to that element
5. collection vs array
   1. array accessed by content not index
6. collection vs stack/queue
   1. order of arrival has no consequence to order of removal
7. Linked based collection implementation
   1. Add to the front so similar to a stack keep pushing objects down and adding on top
8. content must be comparable
9. Equality
   1. == compares the values stored at the variable
      1. Compares values for primitive types
      2. Compares pointers for objects
      3. .equals() is for strings – for objects works like ==
   2. Can override this method to define equality however we like for any obj
      1. So an object does not need to match in every possible class variable
         1. Searching for the comp scientists only by name but getting all info
   3. Obj.compareTo(compareObj)
      1. returns 0 if they are equal
      2. negative if obj < compareObj
      3. positive if obj > compareObj
      4. can use this method to sort a collection
         1. increases cost of inserting and deleting – cant just take last element and put in its spot, also cant just insert at the first null element
            1. have to now shift all elements over
         2. decreases cost of find()
            1. can now use binary search instead of iterative
         3. if you are constantly adding to a collection use unsorted
         4. if you are constantly calling find then use sorted – word density
      5. equivalence
         1. 3 criteria
            1. Reflexive – a=a
            2. Symmetric – x=y, y=x
            3. Transitive – a=b, b=c, a=c

Chapter 6 Lists

1. Maintains linear ordering between elements
2. Can implement all other ADT’s we’ve used
3. Unbounded
4. Duplicates allowed
5. No null elements
6. Add() and set() are optional
   1. Need to be implemented still for exceptions
7. Iterators
   1. Iterates over the content of a list
   2. Two interfaces required
      1. Iterable
      2. Iterator
         1. 3 methods within
            1. Next()
            2. hasNext()
            3. remove()
            4. created and returns a single object Iterator that has the three above methods – done so that way the iterator method can return it.

object is created and set to the beginning of the list

repeated calls on next will return the elements one by one.

hasNext returns true if iterator hasn’t reached the end

remove removes the most recently visited elemtn – to allow for efficient removal if required

Iterator < String > iter = strings.iterator();

String hold;

while (iter.hasNext())

{ hold = iter.next();

if (hold.equals(" gamma"))

iter.remove();

Else

System.out.println( hold);

* + 1. Loop syntax
       1. For(String temp : strings)
          1. System.out.println(temp)

1. ListInterface
   1. Adds index-related operations to collections
      1. 5 new methods
         1. Add()
         2. Set()
            1. Replaces element at the position specified In index with passed element and returns the replaced element
         3. Get()
         4. IndexOf()
         5. Remove()
         6. All of these methods require an index as either input or output
            1. Means exceptions are back despite add and remove returning booleans
2. List implemented first with an array
   1. Elements are stored in an internal array – numElements tracks size
      1. Add elements at index numElements (since index starts at 0)
   2. 3 groups
      1. Collection methods
      2. New list methods
      3. Iterator()
3. Efficiency
   1. Collection methods match array implementation
      1. Except for adding at an index
   2. Index observers run O(I)
   3. Index modifiers run O(N)
   4. Iterator methods run in O(N)?
4. Linked List implementation
   1. Similar to previous data structures
   2. Modifications to collections methods
      1. Simplified modifiers at cost of less efficient observers
      2. Add now works with new rear pointer instead of the head
      3. Targetindex variable is set by the find method
      4. Remove sometimes has to update the rear pointer
   3. Index operations
      1. Has to iterate to index given – O(N)
      2. Eliminates the need to shift the elements
   4. Iterator changes
      1. Iterator obj only moves forwards – not doubly linked
      2. Next – when instantiated set the nextPos to the front
         1. After it is invoked it stores and returns front
         2. Then it moves nextPos to front.getLink()
         3. Currentpos is changed to front
      3. Remove
         1. Since both current and previous pointers are maintained can easily remove
         2. Just set the previous pointer to the nextPos pointer and skip over the node you want to remove
5. List variations
   1. Sorted lists
      1. Do not allow set() and add()
      2. Allows use of compareTo()
   2. Linked lists inside of arrays
      1. Store an array of nodes
      2. Instead of a pointer to a node .next has the index of the next node

Chapter 7 Trees

1. Binary Search Trees
   1. Linked data structure with faster searching
   2. Nonlinear construction
   3. Two basic traversal methods
      1. Breadth first
         1. Process root first then all of its children
         2. In order process all of the roots grand children
         3. After processing the root add all of the children to a data structure and process in that order (Queue LIFO)
2. Machine generated alternative text:
   BFS Illustration 
   root 
   10 
   Villanova University Department of Computing Sciences 
   * 1. Depth first
        1. Process farthest node from the root on the left
        2. When we hit a leaf backtrack
        3. Store the nodes going down and backtrack in reverse order
           1. Stack
3. Machine generated alternative text:
   DFS Illustration 
   root 
   10 
   Villanova University Department of Computing Sciences 
4. BST’s
   1. Nodes can have max 2 children
   2. 2^N-1= max number of nodes you can have
   3. LogN + 1 = number of levels you must have is N = # nodes
   4. Since all nodes in left subtree < root and all nodes in right subtree > root
      1. Allows for searching
   5. 3 new Traversal patterns
      1. Preorder
         1. Root then the left and right subtrees
      2. Inorder
         1. Left subtree first, then root, then right subtree
      3. Postorder
         1. Left and right subtrees first
         2. Then root last
   6. BST methods
      1. Can determine what order you would like your iterator to use
      2. Min() – left most leaf
      3. Max() – right most leaf
      4. isFull – never full
      5. ifEmpty() – if root = null or size = 0
      6. GetIterator()
         1. Change traversal order
      7. Iterator
         1. No longer has remove method
      8. Nodes
         1. Similar to linked list nodes
         2. Uses left and right instead of back/forward
      9. Need a comaparator
         1. To make sure BST implemented correctly
      10. Recursive method size
          1. Could be done iteratively with a count in add / remove
          2. Public method size calling recursive method recSize passing it the root
          3. recSize
             1. general case – call recSize recursively passing in the root
             2. base case – determining if a node is a leaf null by checking if it has children
             3. code works by returning 1 if node is a leaf
             4. and if not it returns 1 + recursively recSize(node.getLeft()) + recSize(node.getRight());
      11. iterative size()
          1. create your own stack
          2. add nodes to the stack and count as they are popped off
          3. similar to tree traversal algorithms
      12. Contains()
          1. Two base cases
             1. If is a leaf return false
             2. And if the nodes comparTo == 0 then return true
          2. Two recursive methods
             1. One with search continuing left and another continuing right
             2. Using compareTo to determine which way to go, if what youre searching for is greater then it goes right – if it is less than it goes left
      13. Iterator
          1. Allow observation of all data in specified order
          2. Standard iterator() method and for-each loop will use inorder traversal
             1. Use getIterator to specify post or pre order
          3. Stores a snapshot of the tree as a queue then perform iterator operations on the queue
             1. Next() hasnext()
             2. Remove is pointless because it’s a snapshot doesn’t do anything
          4. Recursive iterator
             1. Inorder

First call inorder recursively to getLeft() subtree of root and enqueue to the queue

Then call inorder recursively on the right subtree of the root

inOrder( node.getLeft(), q);

q.enqueue( node.getInfo());

inOrder( node.getRight(), q);

* + - * 1. Preorder

If(node!=null)

q.enqueue( node.getInfo());

preOrder( node.getLeft(), q);

preOrder( node.getRight(), q);

* + - * 1. PostOrder

postOrder( node.getLeft(), q);

postOrder( node.getRight(), q);

q.enqueue( node.getInfo());

* + 1. Collection methods in BST
       1. Add()
          1. Recursive implementation
          2. Search for next empty child along the path starting from root and maintaining the comparator
          3. Add Returns a Boolean determining success or failure – since unbounded always true

Rec add returns a BSTNode – a reference to a new tree with the new element

It checks the value against the root then determines right or left – since every subtree is a tree itself with its own root it uses the root.right/left and reads that recursively as the new root into the method and runs again until it hits a leaf aka subtree being searched = null

* + - 1. Remove
         1. Restructure tree to eliminate object
         2. Recursive search to see if its there
         3. 3 cases

No children – make its parent point to null instead of node want to remove

One child – make the parent of the removal point to the child

2 children – find the biggest node in the left subtree of the node you want to remove and place it at spot you are removing the node

Run get max on the subtree and overwrite the node youre replacing with the data from that node (.getData())

* + - * 1. Efficiency

Get/contains – O(Log2N)

Add/remove – O(Log2N)

Whether or not the tree is balanced determines these results