MEMORY, POINTERS, LINKEDLIST (slides)

LinkedList API perspective

- implements List, has the same methods
- reference (pointer) to the first node in a list connected by a reference (pointer) to the next node
- no constant time access to nodes in the middle
    - get() ➝ O(n) for singly linked, O(min(k, size-k)) for doubly (go backward or forward)
    - to get every element one at a time is $O(n^2)$
    - better to use iterator or while hasNext() ➝ O(n)
- removing from the front is much more efficient than ArrayList
    - update node connections ➝ O(1)

memory and references (generally)

- variables for reference types (non primitive) store the location of an object in memory
    - can have multiple references to the same object in memory
- references are copied, changes persist outside of a method
    - vs primitive values are copied, changes won't persist outside of a method
    - still can't "lose" a reference inside a method because the reference is copied
- null reference/pointer
    - null: default value for an uninitialized object
    - check == null

LINKED LIST IMPLEMENTATION & POINTER PROBLEMS (slides, slides)

linked list nodes

- each node has a reference (pointer, memory location) to another node
- calling new Node(...) always creates a Node in memory that didn't exist before
- node.next = otherNode makes node ➝ otherNode
    - next returns the node, not the value
- if node is null (uninitialized), node.next or node.info gives an error
    - no error to get a null reference, yes error to call something on a null reference
- the variable for the "linked list itself" is a reference to the first ListNode

methods

- add() and remove() at front are O(1)
    - removing first ➝ save reference to second element, set first node to null (break connection), set first node to second element
- get() ➝ loops through each element, checks for null for indexOutOfBounds, O(N)
- contains() ➝ loops through list until you find it, O(N)
- traverse with a while loop (list != null)

pointer problems

- append linked lists ➝ O(1), no copying values, just changing pointers
- get last node ➝ go to next until list.next == null

ex: reverse a LinkedList

- temp = list.next list.next = rev rev = list list = temp (rev is reversed so far)

RECURSION (slides)

recursion

- base case: easy answer with small input
- recursive calls: get answer on subset of input
- do something with the result of recursive calls and return
- method does not call itself ➝ calls identical clone with its own state, methods/calls are stacked
- call stack (methods are in a queue)
    - each method call gets it own frame (local vars etc)
    - invoking method does not continue until invoked method returns

random note: invariant

- invariant: what's true for each loop, may become false partway through loop, re-established before guard check

ex: copying a linked list

- iterative ➝ initialize first, call new and link, advance pointers (traverse front to back)
- recursive ➝ create one node that is linked to copy of rest, base case is null

developing and verifying code

- verify base case ➝ always null, sometimes one node
- trace through and ensure result of call is used with small size, generalize from n nodes

ex: recursive reverse

- base case is if list or list.next is null
- store recursive call on list.next
- set list.next.next to list, set list.next to null
- return stored recursive call
- runtime = O(N) ➝ will not have to be able to solve and simplify down
    - T(N) = T(g(N) = N-1)) + (f(N) = O(1))

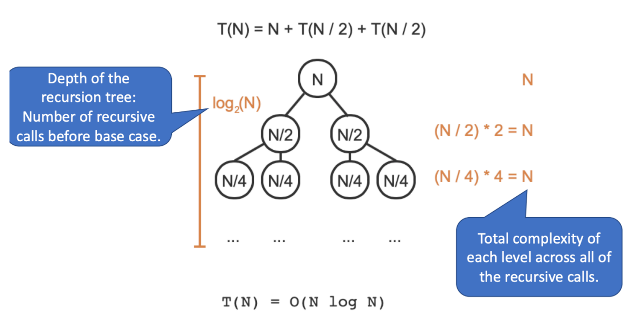
recursive runtime

- $T(N) = a * T(g(N)) + f(N)$
    - $T(N)$: runtime of method with input size N
    - a: number of recursive calls
    - $g(N)$: how much input size decreases on each recursion
    - $f(N)$: runtime of non-recursive code on N
-

Recurrence	Algorithm	Solution
`T(n) = T(n/2) + O(1)`	binary search	`O(log n)`
`T(n) = T(n-1) + O(1)`	sequential search	`O(n)`
`T(n) = 2T(n/2) + O(1)`	tree traversal	`O(n)`
`T(n) = T(n/2) + O(n)`	qsort partition ,find k^{th}	`O(n)`
`T(n) = 2T(n/2) + O(n)`	mergesort, quicksort	`O(n log n)`
`T(n) = T(n-1) + O(n)`	selection or bubble sort	$O(n^2)$

mergesort

- insertion sort without recursion → $O(N^2)$
    - loop through original list, maintain a new sorted list, insert one value at a time in order (looping through new sorted list)
- base case → size 1, return list
- recursive case → mergesort(first half), mergesort(second half), return merged sorted halves with helper merge(first, second)
    - merge() → add to a new list, traverse listA and listB with 2 indices with looping, compare the value at each of 2 indices and add the smaller
- O(N log(N)) → halves at each level so O(log(N)) levels, merge at each level with O(N)
    - merge() → O(nA + nB) = linear



    - mergeSort() →

SORTING, COMPARABLE, COMPARATOR (slides)

Java API sort algorithms

- Collections.sort for List, Arrays.sort for an Array
    - will actually alter the object, even just called within a method
- both implement Timsort (variant of Mergesort)
    - O(N log(N)), nearly linear
    - sorts in place, mutates input instead of making new List/Array
    - stable: does not reorder elements if not needed (ex: 2 elements are equal)

Comparable

- class objects that implement Comparable interface can be sorted → have a naturalOrder
- compareTo()
    - < 0 if this comes before parameter, 0 if equal, > 0 if this comes after parameter
    - ^ if it’s in the right order, return negative
    - method is in class of which this object is an instance
- Strings
    - compareTo() for natural lexicographic ordering (dictionary order, starts at first char)
    - ex: “a”.compareTo(“b”); → -1
- can implement Comparable interface and compareTo()
    - defines a natural ordering, can sort (Collections.sort, Arrays.sort)

Comparator

- use Comparator when not changing the object itself to compare non natural order
    - comparing(Comparator.naturalOrder()) for natural order
- Comparator c, c.compare(a,b)
    - < 0 if a comes before b, 0 if equal, > 0 if a comes after b
    - method is part of Comparator

- ex: sort by length

```java
copy = Arrays.copyOf(a, a.length);
Arrays.sort(copy, Comparator.comparing(String::length));
```

- ex: Collections.sort(schools, Comparator.comparing(University::getName));
    - ^no () because passing in the getName function name itself, not the return value
- to combine sequence of comparison → Comparator.comparing(_::_).thenComparing
- Comparator with lambdas (for when you compare something without a getter)
    - Comparator<Object type> c = (given a,b) -> (what c.compare(a,b) should return)
    - ex: sort by first elements

```java
Comparator<int[]> comp =
                    (a, b) -> (a[0] - b[0]);
```

Type we want to compare

Given an a and a b of that type...

comp.compare(a,b) should return this expression

    - can use sort(_, createdComparator)

runtime complexity of sort() and Comparator → O(N log(N))

- Arrays.sort() and Collections.sort() call either compareTo() (default) or compare() (if Comparator)

comparisons in mergesort

- one comparison per loop iteration / per element merged
- O(CN log(N)), where C is complexity of compareTo() or compare()
- C is not constant

```java
public class ListComp implements Comparator<List<Integer>> {
    @Override
    public int compare(List<Integer> list1, List<Integer> list2) {
        int minLength = Math.min(list1.size(), list2.size());
        for (int i=0; i<minLength; i++) {
            int diff = list1.get(i) - list2.get(i);
            if (diff != 0) {
                return diff;
            }
        }
        return 0;
```

Runtime complexity of this Comparator may depend on the length of the two Lists being compared.

    - worst case to sort here is O(MN log(N)) (N ArrayLists, each with M elements)

- M is complexity of each call of compare()

binary search

- given a sorted list of N elements and a target, return index i of target or -1 if not in list
- O(log(N)) → cut down search space by half at each step
- process
    - low (initially 0) and high (initially N-1) mark the limits of active search space
    - loop while (low = high) → while there's anything left to search
        - loop invariant → if target is in array/list, it is in [low, high]
    - compare mid ((low+high)/2) to target
    - search in lower (high = mid-1) or upper half (low = mid+1)
        - not high = mid or low = mid to prevent infinite loop in edge cases → you could get stuck at just a few elements where the range won't decrease
- does not guarantee the first or last index of a match if there are multiple