Final Exam CHEAT SHEET

ArrayList
- O(n) add

X2 LL
- Remove end
  ↳ go bkwds.
  set pointers
  O(2+n)

comparing
- move ←
+ move →
0 stay

comparator	comparable
object	natural order
specific order	implementation

Tree Traversal

Pre: PLR (read/write)
In: LPR (print)
Post: LRP (destroy)

	Insert (B/W)	Get Min (peek/pop)
unsorted vector	O(1)	O(n)
sorted vector	O(log n + n)/O(n)	O(1)
Heap	O(log n)	O(1)/O(log n)
Bal. BST	O(log n)	O(1)/O(log n)

Big Oh Table

PQ: Search O(N); Insert O(1)/O(log N); Delete O(N); FindMin O(1)
Trie O(w) for all
TreeSet O(log N) for all
HashSet

makeTree (in → HuffNode)
```
int[] f = count(in) //arr char f
PQ<HN> pq
for(i=0→ALPH_SIZE)
    if (f[i]!=0) pq.add(n HN(i, f[i]))
pq.add(n HN(PSEUDO_EOF, 0))
w(pq.size > 1)
    HN a = pq.poll
    HN b = pq.poll
    HN c = n HN(-1, a.wt + b.wt, a, b)
    pq.add(c)
r pq.poll
```

extract (HN curr, S path)
```
if (isLeaf(curr))
    array[curr.value] = path
    return
extract(curr.left, path+0)
extract(curr.right, path+1)
```

writeHeader (HN curr, out)
```
if (isLeaf(curr))
    out.writeBits(1,1)
    out.writeBits(9, curr.value)
    return
out.writeBits(1,0)
writeHeader(curr.left, out)
writeHeader(curr.right, out)
```

Huff Node impl. comparator
value, weight, left, right
return myWeight − other.myWeight

Huff Decoding
```
(S arch, S[] dict) → S
Map<S→S> put(dict[i], ...
character.toS((char)(i+65)))
out = newSB
int start=0, end=1
while (end<arch.l+1){
    S str = arch.subS(start, end)
    if (map.containsKey(str)
        out.append(map.get(str))
        start = end
        end = start + 1
    else end++ }
r out.toS
```

writeBody (in, out)
```
w(true)
    int i = in.readBits(BITS_PER_WORD)
    if (i==-1) break
    S code = array[i]
    out.writeBits(code.l, I.parseInt(code, 2))
S str = array[PSEUDO_EOF]
out.writeBits(str.l, I.parseInt(str, 2))
```

compress (in, out)
```
HN huff = makeTree(in)
array = n S[ALPH_SIZE+1]
extract(huff, "")
out.writeBits(B_P_I, HUFF #)
writeHeader(huff, out)
writeBody(in, out)
```

readHeader (in) → HN
```
if (in.rB(1)==0)
    HN left = readHeader(in)
    HN right = same↑
    r n HN(0, 0, left, right)
e r n HN(in.rB(9), 0)
```

parseBody (HN curr, in, out)
```
HN root = curr
w(T)
    int i = in.rB(1)
    if (i==-1) break
    if (i==1) curr = curr.right
    else curr = curr.left ← if 0
    if (isLeaf(curr))
        if curr.value == PSEUDO_EOF
            return
        else {
            out.wB(8, curr.value)
            curr = root
throw n HuffException
```

decompress (in, out)
```
if (in.rB(B_P_I) != HUFF#)
    throw n HuffException
HN curr = readHeader(in)
parseBody(current, in, out)
```

Map Iterate
```
for(Map.Entry<S,I> entry: m.entrySet())
```

Scanner Tricks
```
w(in.hasNext())
    S str = in.nextLine().replace(":","","")
    S[] words = str.split(", ")
    in.close()
```

* Make sure to init maps + check contains

Erdos Number

```
HM vert <S -> Coll<S>>
HM erdos <S, I>

addToMap(S[] a, HM M)
Set<S> t = n HS<S>(A.asL(a))
for(S s:t)
  if(M.containsKey(s))
    M.get(s).addAll(t)
    M.get(s).remove(s)
  else
    ~same ↑
    (map: name -> collab)

getAdjList (S[] pubs) -> HM<S, Coll<S>>
HM<S, Coll<S>> M
for(i=0 -> pubs.l)
  S[] names= pubs[i].split(" ")
  addToMap(names, M)
return M

bfs(S start) -> TS<S>
erdos= n HM<S, I>
erdos.put(start,0) // base
TS<S> visited = n TS<S>
Q<S> qu =n LL<S>
visited.add(start)
qu.add(start)
w(qu.size >0)
  S v= qu.remove
  for(S adj : vert.get(v))
    if(!visited.contains(adj))
      visited.add(adj)
      qu.add(adj)
      erdos.put(adj, erdos.get(v)+1)
return visited

calcNumbers (S[] pubs)
vert = getAdjList(pubs)
Set<S> noE = vert.keySet
TS<S> myV = bfs("ERDOS")
noE.removeAll(myV)
TS<S>out= n TS<S>
for(S s:myV)
  s=s+" "+myE.get(s)
  out.add(s)
TS<S>ret
ret.addAll(out)
ret.addAll(noE)
return ret.toA(n S[out.size])
```

Insertion

- insert unsort → sorted list

Selection Sort

- Div. into sort/unsort regions
- Move max unsort to sorted
- Heap sort is improved version

Quick Sort

- Used for primitive []
- Choose pivot → split vector
- Very fast when fine-tuned

MergeSort

- Used for Objects
- Recursive (split)
- Good for // (work on dif. sections)

Array O(1) access, O(N) everything else
Stack O(N) access/search, O(1) insert/delete
LLx1 O(N) access/search, O(1) insert/delete (unsorted)
LLx2 Same as above
HashTable O(1) best, O(N) worst
BST O(log(n)) balanced, O(N) unbalanced
Binary Heap O(1) find max, O(logN) insert/delete/poll

Adj. List O(V+E) storage, O(1) add V/E worst space
QuickSort O(NlogN) best/avg. $O(N^2)$ worst, O(logN)
MergeSort O(NlogN) all, O(N) worst space
HeapSort O(NlogN) all, O(1) worst space
InsertionSort O(N) best, $O(N^2)$ avg/worst, O(1) worst space
SelectionSort $O(N^2)$ all, O(1) worst space

```
ith (LN list, int i) -> LN
for(k=0-> i && list≠∅)
  list= list.next
r list

subseq(LN list, int begin, int end)
+1 if(list=∅) r ∅
+2 LN splice = ith(list,begin-1)
   if(splice=∅) r splice
   LN result= splice.next
   LN resultTail = ith(result,end-begin-1)
   if(resultTail≠∅)
     splice.next= resultTail.next
     resultTail.next= ∅
   else //end=l-1
     splice.next=∅
   r result
```

Exam Two Cheat Sheet

by Brian Lin

Comparisons

Array[]	vs.	ArrayList<String>
new String[n]		new ArrayList<String>()
.length		.size()
[k]		.get(k)
Arrays.sort(w)		Collections.sort(~)

Comparable	vs.	Comparator
compareTo method		Compare method
· "natural" order		· Mult. sorting methods
· ⊖ if this < o2		· 2 arguments
· 0 if this == o2		
· ⊕ if this > o2		

Path Exists

```
// check out of bounds
if (isGoal(row, col)) r T
if (isVisited(row,col)) r F
markVisited(row, col)
r (!isBlocked(row, col, row+1, col)
    && pathExists(row+1, col)) || ...
```

Conversions

```
String → String[]
  s.split("\\s+");
String[] to Set/List
List<String> myList = Arrays.asList(sA);

List/ArrayList → Set                (for a set)
Set<String> set1 = new TreeSet<String>(list);

Set/List → String[]
String[] sA = mySet.toArray(new String[mySet.size()]);
                            (myList)        (myList)
Coll → String
String out = new String();
for (String elem : input)
    out = out + " " + elem;
return out.trim();
```

Forbes' Pro Strats

Reverse Sort → `Arrays.sort(a, Collections.reverseOrder())`

`Collections.reverse(arrList);`

`TreeSet<String>((s1,s2) -> -s1.compareTo(s2));`

Big O

N=100 → t = .5 ms

N=500 → t = ? using nlogn

$.5 = k(100 \log_2 100)$

$? = k(500 \log_2 500)$

Recurrence Relations

$T(N) = T(?) + O(?)$

Previous recursive step — Other operations

Balanced: $T(n/2)$ $T(n/2)$ $= 2T(n/2)$

Unbalanced: $T(n-1)$ $T(1)$ $= T(n-1) + T(1)$

Tries

```
add(S w)
  N curr = root
  for(i=0 → w.l)
    if(curr.children.contains(
    w.charAt(i)))
      curr = curr.children.
        get(charAt(i))
    else curr.children.put(
        w.charAt(i), n N())
    curr = curr.children.get(...)
  curr.isWord = T
union(N l, N r)
  if (l == ∅) r copyTrie(r)
  else if (r == ∅) r copyTrie(l)
  else {
    N curr = n N()
    bool hasBelow = F
    // get set of all children
    HS<ch> allC = n HS<ch>(l.chil.keySet())
     allC.addAll(r.chil.keyset())
    for (char ch: allC)
        N uChil = trieUnion(l.chil.get(ch),
                            r.chil.get(ch))
        if(uChil ≠ ∅) //something found
          hasBelow = T
          curr.chil.put(ch, uChil)
    curr.isWord = l.isWord || r.isWord // in union?
    if (hasBelow || curr.isWord) //something to r
        r curr
    else r ∅
intersection  addAll/retainAll
              curr.isWord = l.isWord && r.isWord
```

BSAutoComplete

```
firstIndexOf(Term[] a, Term key, Comp)
  i low=-1
  i mid=0
  i high=a.l-1
  c = comp.compare(a[mid], key)
  w(h-l>1)
    m=(l+h)/2
    if(c>=0) h=m
    if(c<0) l=m
  if(comp.compare(a[(l+h)/2],key)==0)
    r (h+l)/2
  if(comp.compare(a[(l+h)/2+1],key)==0)
    r (h+l)/2+1
  else r -1
S[] topKMatches(S pref, i k)
    If(pre==∅) error
    PQ<T> pq = n PQ<T>...
    (k, n T.weightOrder())
    i lst = firstIdx(_terms, ...
    n T.PrefixOrder(pre.l))
    i last = same
    if(lst==-1) r n S[0]
    T[] t = nT[last-lst+1]
    f(i=lst, ind=0 → i<t.l)
      if(pq.size()<k)
        pq.add(t)
      else if(pq.peek().getW ...
      <t.getW
        pq.remove()
        pq.add(t)
    i numRes = Math.min(k, pq.size)
    S[] ret = n S[numRes]
    for(i=0 → numRes)
      ret[numRes-1-i] = pq.remove()...
      .getWord()
    r ret
```

DrawTree

```
pr i root
L<N> n

pr cl N
  pr S nm
  pr L<N> chil
  pb N(S s)
    nm = s
    chil = n AL<N>

pb S[] draw(i[] par, S[] names)
  n = n AL<N>()
  for(i=0 → par.l)
    n.add(n N(names[i]))
  for(i=0 → par.l)
    if(par[i]==-1) root=i
    else n.get(par[i]).chil...
    add(n.get(i))
  L<S> out = draw(n.get(root), F)
  r out.toArray(n S[par.l])

pr bool isLast(N n, i i)
  if(n.chil.size()-1==i) r T
  else ret F

pr L<S> draw(N n, bool drawBar)
  L<S> l = n AL<S>()

  n.nm = "+-" + n.nm
  S str
  if(drawBar) str = "|  "
  else str = "   "

  l.add(n.nm)

  for(i=0 → n.chil.size())
    L<S> cList = draw...
    (n.chil.get(i), !isLast(n,i))

    for(S s: cList)
      s = str+s
      l.add(s)

  r l
```

DNA LinkStrand

```
pb c| Node
  $ info
  Node next
  Node ($ s)
      info = s
      next = ∅
pb LinkStrand ($ s)
  mySize = s.ℓ
  myAppends = 0
  myFst = new N (s)
  myLast = myFst
```

```
$ toS()
  N curr = myFst
  SB out = new SB()
  w (curr ≠ ∅)
     out.app(curr.info)
     curr = curr.next
  return out.toS
```

```
reverse()
  LS out = new LS()
  Stack<$$> myStack = n Stack<$>
  N curr = myFst
  Map<$,$$> myMap = n HMap<$,$$>()
  w (curr ≠ ∅) // Reverse $
     SB my$B = new SB (curr.info)
     if (!myMap.containsKey(curr.info))
        myMap.put (curr.info, my$B.reverse().to$())
     curr.info = myMap.get (curr.info)
     myStack.push (curr.info)
     curr = curr.next

  // Pop $ out of stack
  bool fst = T
  w (!myStack.isEmpty())
     if (fst)
        out = n LS (myStack.pop())
        fst = F
     else out.app(myStack.pop())
  return out
```

```
cutAndSplice ($ enz, $ spl)
  if (myFst.next != ∅)
     throw error → LS has > 1 nodes
  i pos, start = 0
  SB search = new SB (myFst.info)
  bool first = ⊥
  LinkStrand ret = ∅
  while ((pos = search.indexOf(enz, pos)) >= 0)
     if (first)
        ret = new LS (search.substring(start, pos))
        first = F
     else ret.append(search.substring(start, pos))
     start = pos + enz.ℓ
     ret.append(spl)
     pos++

  if (start < search.ℓ) //left never found
     if (ret == ∅) ret = new LS("")
     else ret.append(search.sub$(start))
```

```
append (IDnaStrand dna)
  if (dna instanceof LS)
     LS myStrand = (LS) dna
     //old last points to new fst
     myLast.next = myStrand.myFst
     // New last → full last
     myLast = myStrand.myLast
     mySize += myStrand.mySize
     myAppends++
     return this
  else return append (dna.toString)

append ($ dna)
  N myNode = new N (dna)
  myLast.next = myNode
  myLast = myNode
  mySize += dna.length()
  myAppends++
  return this
```

```
SB
  append(?)
  charAt(i)
  delete(i1, i2)
  insert(i, ?)
  length()
  toString()
  reverse()
```

BST count

```
HM<I, Long> m
pb long howMany(i[] a)
  m = n HM<save>
  r howMany(a.ℓ)
pb long howMany(i size)
  if (size == 0)
     r 1
  if (m.containsKey(size))
     r m.get(size)
  long trees = 0
  for (i left=0 → size-1)
     i right = size-1-left
     trees += howMany(left) *
              howMany(right)
  m.put(size, trees)
  r trees
```

PercolationDFS

```
pb i[][] myGrid
// init stuff (not shown)
open(i i, i j)
  i n = myGrid.ℓ
  if (i<0||j<0||i≥n||j≥n)
     // out of bounds
  for (k=0 → n) //flush
     for (h=0 → n)
        if (myGrid[k][h]==FULL)
           myGrid[k][h] = OPEN
  myGrid[i][j] = OPEN
  for (k=0 → myGrid[0].ℓ)
     dfs(0,k)

bool percolates()
  bool out = F
  for (i ind: myGrid[n-1])
     if (ind==FULL)
        out = T
  return out

dfs (i row, i col)
  if (row<0||col<0||row≥n||col≥n)
     return
  if (!isOpen(row,col)) return
  if (isFull(row, col)) return

  myGrid[row][col] = FULL
  dfs(row±1, col±1)
```

PercolationUF

```
getIndex(i row, i col)
  //check out of bounds
  return row*N + col

open(i i, i j)
  myGrid[i][j] = OPEN
  if (i==0)
     myUnion.union(getI(i,j), sourceI)
  if (i==N-1)
     myUnion.union(getI(i,j), sinkI)
  connect(i,j)

isFull(i i, i j)
  if (isOpen(i,j) &&...
  myUnion.connected(getI(i,j), sourceI)
     return T
  else return F

percolates()
  return myUnion.connected(sourceI, sinkI)

connect(i row, i col)
  //check out-of-bounds
  if (row>0 && isOpen(row-1,col)
     myUnion.union(getI(row,col), getI(row-1,col))
  //repeat for other directions
```

TrieAutocomplete

```
add($ s, double wt)
  N curr = myRoot
  for (i=0 → s.ℓ)
     char ch = s.charAt(i)
     if (curr.mySTMW < wt)
        curr.mySTMW = wt
     if (!curr.chil.containsKey(ch))
        curr.chil.put(ch, n N...
        (ch, curr, wt))
     curr = curr.chil.get(ch)
  // now you are @ a word node!
  double oldWt = curr.myWeight
  curr.myWeight = wt
  curr.isWord = T
  curr.myWord = s

  if (curr.mySTMW < wt)
     curr.mySTMW = wt
  if (oldWt > wt)
     w(curr ≠ ∅)
        if(curr.par.mySTMW == oldWt)
           double max = Math.max(...
              .curr.par.myWt
              .wt
              .curr.par.my$TMW)
           curr.par.mySTMW = max
        curr = curr.par
```

```
topKMatches ($ pre, i k)
  PQ<N> pq = n PQ<ND>(k, n RevSTMW)
  L<N> ℓ = n AL<N>()
  for (i=0 → pre.ℓ)
     char ch = pre.charAt(i)
     if (curr.chil.get(ch) ≠ ∅)
        curr = curr.chil.get(ch)
     else r n $[]
  pq.add(curr)
  w(!pq.isEmpty())
     if (ℓ.size ≥ k)
        if (ℓ.get(0).myWeight/wt...
           <pq.peek().mySTMW)
              if(pq.peek().isWord)
                 ℓ.remove(ℓ.size-1)
        else break
     curr = pq.poll()
     if (curr.isWord) ℓ.add(curr)
     pq.addAll(curr.chil.values())
  L<$$> ret = n AL<$$>()
  Col.sort(out, n MWC)
  i stop = 0
  for (N n: out)
     ret.add(n.myWord)
     stop++
     if(stop≥k) break
  r ret.toArray(n $[Math.min(k, ret.size)])

pb static class MWC imp Comp<N>
  pb i compare(N o1, N o2)
     r -Double.compare(o1.myWt, o2.myWt)
```

PriorityQueue

```
add
contains
remove
Poll
peek
size
toArray
```

Arrays

```
asList()
Sort()
toString()
```

Methods

Integer

```
intValue()
equals(o)
compareTo(I)
valueOf(?)
```

String

```
charAt(i)
compareTo(S)
concat(S)
contains(S)
equals(o)
hashCode()
indexOf(c)
length()
replace(c1, c2)
split(regex)
substring(i1, i2)
trim()
valueOf(o)
```

Map

```
entrySet()
containsKey/Value
entrySet()
get(key)
keySet()
put(k,v)
remove(k)
size()
values()
```

List

```
add(e)
addAll(coll)
contains(o)
get(i)
indexOf(o)
remove(o)
retainAll()
size()
sort(?)
toString()
```

UnMemo

$T(N) = 2T(N-1) + 2T(N-2) + \dots + 2T(0) + O(N)$

Memo

$T(N) = T(N-1) + O(N)$