

```

1
2 =====
3         STACKS
4 ===== //all interfaces allow these interactions with stacks:
5     create an empty stack
6     push an item onto the stack //items are always added to and removed from the TOP of
    the stack
7     pop an item off the stack
8     look at the top item of the stack
9     is the stack empty?
10 ----- //some interfaces let you do these:
11     how many items are on the stack?
12     look at any item on the stack
13
14 #include <stack>
15 using namespace std;
16
17 int main()
18 {
19     stack<int> s;
20     s.push(10);
21     s.push(20);
22     if (!s.empty())
23         cout << s.size(); //size is 2
24     s.pop(); //pop an item off the top
25     int n = s.top(); //n is the top item on the stack, which is 10 here
26
27 }
28
29 //Maze function using stacks
30
31 class Coord
32 {
33 public:
34     Coord(int rr, int cc) : m_r(rr), m_c(cc) {}
35     int r() const { return m_r; }
36     int c() const { return m_c; }
37 private:
38     int m_r;
39     int m_c;
40 };
41
42 bool pathExists(char maze[][10], int sr, int sc, int er, int ec)
43 {
44     stack<Coord> coordStack;
45     Coord start(sr, sc);
46     Coord end(er, ec);
47
48     coordStack.push(start);
49     maze[start.r()][start.c()] = '#';
50
51     while (!coordStack.empty())
52     {
53         Coord loc = coordStack.top();
54         coordStack.pop();
55
56         if ((loc.r() == end.r()) && (loc.c() == end.c()))
57             return true;
58
59         if (maze[loc.r() + SOUTH][loc.c()] == '.') //code puts all empty spaces around
    location onto the stack, and eventually checks them all, resulting in every
    accessible maze location being checked
60     {
61         Coord newloc(loc.r() + SOUTH, loc.c());
62         maze[newloc.r()][newloc.c()] = '#';
63         coordStack.push(newloc);
64     }
65     if (maze[loc.r()][loc.c() + WEST] == '.')
66     {

```

```

67         Coord newloc(loc.r(), loc.c() + WEST);
68         maze[newloc.r()][newloc.c()] = '#';
69         coordStack.push(newloc);
70     }
71     if (maze[loc.r() + NORTH][loc.c()] == '.')
72     {
73         Coord newloc(loc.r() + NORTH, loc.c());
74         maze[newloc.r()][newloc.c()] = '#';
75         coordStack.push(newloc);
76     }
77     if (maze[loc.r()][loc.c() + EAST] == '.')
78     {
79         Coord newloc(loc.r(), loc.c() + EAST);
80         maze[newloc.r()][newloc.c()] = '#';
81         coordStack.push(newloc);
82     }
83     }
84     return false;
85 }
86
87 =====
88 QUEUES //like a stack but backwards
89 ===== //all interfaces allow these interactions with queues
90     create an empty queue
91     enqueue an item onto the queue //items are always added to and removed from the
92     BACK of the queue
93     dequeue an item from the queue
94     look at the front item of the queue
95     is the queue empty?
96     ----- //some interfaces let you do these
97     how many items are in the queue?
98     look at the back item of the queue
99     look at any item in the queue
100
101 have to #include <queue>
102
103 //Maze function using queues
104 bool pathExists(char maze[][10], int sr, int sc, int er, int ec)
105 {
106     queue<Coord> coordQueue;
107     Coord start(sr, sc);
108     Coord end(er, ec);
109
110     coordQueue.push(start);
111     maze[start.r()][start.c()] = '#';
112
113     while (!coordQueue.empty())
114     {
115         Coord loc = coordQueue.front();
116         coordQueue.pop();
117
118         if ((loc.r() == end.r()) && (loc.c() == end.c()))
119             return true;
120
121         if (maze[loc.r() + SOUTH][loc.c()] == '.') //code puts all empty spaces around
122             location onto the stack, and eventually checks them all, resulting in every
123             accessible maze location being checked
124         {
125             Coord newloc(loc.r() + SOUTH, loc.c());
126             maze[newloc.r()][newloc.c()] = '#';
127             coordQueue.push(newloc);
128         }
129         if (maze[loc.r()][loc.c() + WEST] == '.')
130         {
131             Coord newloc(loc.r(), loc.c() + WEST);
132             maze[newloc.r()][newloc.c()] = '#';
133             coordQueue.push(newloc);
134         }
135     }
136 }

```

```

133         if (maze[loc.r() + NORTH][loc.c()] == '.')
134         {
135             Coord newloc(loc.r() + NORTH, loc.c());
136             maze[newloc.r()][newloc.c()] = '#';
137             coordQueue.push(newloc);
138         }
139         if (maze[loc.r()][loc.c() + EAST] == '.')
140         {
141             Coord newloc(loc.r(), loc.c() + EAST);
142             maze[newloc.r()][newloc.c()] = '#';
143             coordQueue.push(newloc);
144         }
145     }
146     return false;
147 }
148
149
150 =====
151 Notations
152 =====
153 prefix notation:
154 f(x,y,z)
155 add(sub(8, div(6,2), 1) //also works: add sub 8 div 6 2 1      also works: + - 8 / 6 2 1
156
157 infix notation:
158 8-6/2+1 //the same thing as the line above, can be more confusing for a human to parse
    sometimes?
159
160 postfix notation:
161 8 6 2 / - 1 + //once again, the same thing
162 ((8 (6 2 /) -) 1 +) //the associated groupings.
163 //Postfix notation doesn't need any additional specificity. To the computer, it's
    always unambiguous what it operates on
164 //postfix is actually easier to process than prefix/infix notation, runs faster than
    both. Common expression evaluation: given something in infix, translate it to postfix,
    then it's easy to evaluate
165
166 8 6 2 / - 1 +
167 // evaluating a postfix sequence: (pseudocode)
168
169 operand stack
170 run through the postfix expression:
171     when you encounter a number: push it onto the stack
172     when you encounter an operand: pop off the top two numbers, run the expression,
        push the result back onto the stack
173 if this is a valid postfix expression, the stack will have exactly one value on it (
    the value of the expression)
174
175
176 // translating infix to postfix:
177
178 make an operator stack and a postfix string
179
180 run through the expression:
181     numbers: push to the postfix string
182     operator:
183         open parens ->always push
184         top of stack is parens -> always push
185         current operator has a higher precedence than top of stack: push
186         close parens: pop the stack to the postfix string until you pop an open
            parens
187 if the current operator is lower precedence than what's on top of the stack, it
    goes lower down
188 //try it with some numbers! - see if it works
189
190 =====
191 Making a picture-drawing algorithm
192
193 class Shape //generalization of the idea of circles and rectangles

```

```

194 {
195     void move(double xnew, double ynew);
196     virtual void draw() const;    //draw is called differently for each type of shape.
197     double m_x;
198     double m_y;
199 };
200
201 class Circle : public Shape //introducing a new type called circle, and it's a kind of
    shape
202 {
203     virtual void draw() const;
204     double m_r; //since circle is a type of shape, you only have to declare member
        functions that are unique to it. it already has move, m_x, m_y, etc. Draw seems to
        be a special case watch out.
205 };
206
207 class Rectangle : public Shape
208 {
209     virtual void draw() const;
210     double m_dx;
211     double m_dy;
212 };
213
214 Shape* pic[100];    //Shape is a heterogeneous collection of items in a strongly-typed
    language
215 pic[0] = new Circle; //circle "is-a-kind-of" // "is-a" shape
216 pic[1] = new Rectangle;
217 pic[2] = new Circle;
218 //data abstraction is pretty powerful
219
220
221 for (int k = 0; k < ...; k++)
222     pic[k]->draw();
223
224 void f(Shape& s)
225 {
226     s.move(10, 20);
227     s.draw();
228 }
229
230 void Shape::move(double xnew, double ynew)
231 {
232     m_x = xnew;
233     m_y = ynew;
234 }
235
236 void Shape::draw() const
237 {
238     ...draw a cloud centered at m_x, m_y...
239 }
240
241 void Circle::draw() const
242 {
243     ...draw a circle...
244 }
245
246 void Rectangle::draw() const
247 {
248     ...draw a rectangle...
249 }
250
251 ==
252 review sesh notes
253
254 member variables are constructed before the constructor is called -> inside out
255 member variables are destructed after the desctructor is called -> outside in
256
257 if member variables are dynamically allocated, they will not be deleted automatically
    when/after the destructor is called - gotta add that to the destructor yourself to get

```

```

it to happen
258
259
260 default copy constructor just copies member variables from the existing object to the
new one
261 you need your own copy constructor when:
262     the object contains a pointer to something in memory. otherwise, the pointers will
        point to the same block of memory, instead of just being a copy. This means that
        when you modify one the other changes too.
        ( ^ dynamic memory allocation)
263
264
265
266
267 =====
268 Inheritance / Virtual Functions
269 =====
270 //if a class is designed to be a base class, declare a virtual destructor and implement
    it
271 // STEPS OF CONSTRUCTING OBJECTS: THE FIRST STEP
272 // Construct the Base Part
273 // Construct the Data Members
274 // Execute the body of the constructor
275
276 STEPS OF DESTRUCTION
277     1. Execute the body of the destructor
278     2. Destroy the data members
279     3. Destroy the base part //new third step
280
281 class Device //device is the base class for several other things.
282 {
283     public:
284     virtual ~Device(){}
285     virtual void open() = 0;
286     virtual void write(char c) = 0;
287     virtual void close() = 0;
288 };
289
290 class BannerDisplay : public Device
291 {
292 public:
293     virtual void open(); //implemented somewhere
294     virtual void write(char c); //implemented somewhere
295     virtual void close(); //implemented somewhere
296 private:
297     ...
298 };
299
300 class Modem : public Device
301 {
302 public:
303     virtual void open(); //implemented somewhere
304     virtual void write(char c); //implemented somewhere
305     virtual void close(); //implemented somewhere
306 private:
307     ...
308 };
309
310 void writeString(Device& d, string msg)
311 {
312     for (int k = 0; k != msg.size(); k++)
313         d.write(msg[k]);
314 }
315
316
317 //program to send warning messages in various mediums
318 enum CallType {
319     VOICE, TEXT
320 };
321

```

```

322 class Medium
323 {
324 public:
325     Medium(string id) : m_id(id) {}
326     virtual ~Medium() {}
327     virtual string connect() const = 0;
328     virtual string transmit(string msg) const { return "text: " + msg; }
329     string id() const { return m_id; }
330 private:
331     string m_id;
332 };
333
334 class Phone : public Medium
335 {
336 public:
337     Phone(string num, CallType type) : Medium(num) { m_type = type; }
338     ~Phone() { cout << "Destroying the phone " << id() << "." << endl; }
339     string connect() const { return "Call"; }
340     string transmit(string msg) const
341     {
342         switch (m_type)
343         {
344             case VOICE:
345                 return "voice: " + msg;
346             case TEXT:
347                 return "text: " + msg;
348         }
349     }
350 private:
351     CallType m_type;
352 };
353
354 class TwitterAccount : public Medium
355 {
356 public:
357     TwitterAccount(string id) : Medium(id) {}
358     ~TwitterAccount() { cout << "Destroying the Twitter account " << id() << "." << endl; }
359     string connect() const { return "Tweet"; }
360 };
361
362 class EmailAccount : public Medium
363 {
364 public:
365     EmailAccount(string id) : Medium(id) {}
366     ~EmailAccount() { cout << "Destroying the email account " << id() << "." << endl; }
367     string connect() const { return "Email"; }
368 };
369
370 void send(const Medium* m, string msg)
371 {
372     cout << m->connect() << " using id " << m->id()
373         << ", sending " << m->transmit(msg) << endl;
374 }
375
376
377 =====
378 RECURSION
379 =====
380
381
382 I want to sort a pile of N items:
383
384 if (N > 1)
385 {
386     split the pile about evenly into two insorted piles
387     sort the left subpile
388     sort the right subpile
389     merge the two sorted subpiles into one sorted pile

```

```

390     }
391     //to debug, assume a recursive function works
392     void sort(int a[], int b, int e)
393     {
394         if (e - b >= 2)
395         {
396             int mid = (b + e) / 2;
397             sort(a, b, mid);
398             sort(a, mid, e);
399             merge(a, b, mid, e);
400         }
401     }
402
403     int main()
404     {
405         int arr[5] = {40, 30, 20, 50, 10};
406         sort(arr, 0, 5);
407     }
408
409
410
411

```

To prove $P(N)$ for all $N \geq 0$:

1. Prove: $P(0)$
2. Prove: If $P(k)$ is true for all $k < N$, then $P(N)$

```

416
417     //Some recursive functions
418
419     bool somePredicate(string s) //predicate to check if the string has things in it.
420     Returns true if the string is NOT empty, and false otherwise
421     {
422         return !s.empty();
423     }
424
425     bool allTrue(const string a[], int n) // Return false if the somePredicate function
426     returns false for at least one of the array elements; return true otherwise.
427     {
428         if (!somePredicate(a[n - 1])) //if the item fails the predicate, return false
429             return false;
430
431         if ((n - 1) > 0) //if it doesn't fail and it's not the last item in the list, check
432         the next one
433         if (allTrue(a, n - 1))
434             return true; //the return must be conditional on allTrue, not just after
435             it, or the function returns true every time
436     }
437
438     int countFalse(const string a[], int n) // Return the number of elements in the array
439     for which the somePredicate function returns false.
440     {
441         int i = 0;
442         if (n > 0)
443         {
444             if (!somePredicate(a[n - 1]))
445                 i += (1 + countFalse(a, n - 1));
446             else
447                 i += (0 + countFalse(a, n - 1));
448         }
449         return i;
450     }
451
452     // Return the subscript of the first element in the array for which
453     // the somePredicate function returns false. If there is no such
454     // element, return -1.
455     int firstFalse(const string a[], int n)
456     {
457         if (n <= 0)
458             return -1;

```

```

454
455     int x = firstFalse(a, n - 1);
456
457     if (!somePredicate(a[n - 1]) && x == -1)
458         return n - 1;
459     return x;
460 }
461
462 // Return the subscript of the least string in the array (i.e.,
463 // return the smallest subscript m such that a[m] <= a[k] for all
464 // k from 0 to n-1). If the function is told to examine no
465 // elements, return -1.
466 int indexOfLeast(const string a[], int n)
467 {
468     if (n <= 0)
469         return -1;
470     int i = 0;
471     if (n - 1 > 0)
472     {
473         int comp = indexOfLeast(a, n - 1);
474         if (a[comp] < a[n - 1])
475             i = comp;
476         else
477             i = n - 1;
478     }
479     return i;
480 }
481
482 // If all n2 elements of a2 appear in the n1 element array a1, in
483 // the same order (though not necessarily consecutively), then
484 // return true; otherwise (i.e., if the array a1 does not include
485 // a2 as a not-necessarily-contiguous subsequence), return false.
486 // (Of course, if a2 is empty (i.e., n2 is 0), return true.)
487
488 bool includes(const string a1[], int n1, const string a2[], int n2)
489 {
490     bool i = false;
491     if (n1 > 0 && n2 > 0)
492     {
493         if (a1[0] == a2[0])
494         {
495             a2++;
496             n2--;
497         }
498         a1++;
499         n1--;
500         i = includes(a1, n1, a2, n2);
501     }
502     if (n2 == 0)
503         return true;
504     else
505         return i;
506 }
507
508
509
510

```

=====

trees?

=====

it's basically just applied math, very little actual programming

N	$2N^2+1000N+1000$	$3N^2-2N+5$	$10000N$
100	123,000	29,805	1,000,000
1000	3,001,000	2,998,005	10,000,000
10000	210,001,000	299,980,005	100,000,000


```

523 100000          20,100,001,000          29,999,800,005          1,000,000,000
524
525 Exponents totally dominate other terms when N is large (duh)
526
527
528 A function  $f(N)$  is  $O(g(N))$  if there exist  $N_0$  and  $k$  such that for all  $N \geq N_0$ ,  $|f(N)| \leq k \cdot g(N)$ 
529
530  $f(N)$  is "order  $g(N)$ "
531
532
533
534 for (int i = 0; i < N; i++) <-----  $O(N)$ 
535     c[i] = a[i] * b[i]; <-----  $O(1)$ 
536
537 for (int i = 0; i < N; i++) <-----  $O(N^2)$  (
since there's 2  $O(N)$  in there)
538 { <-----  $O(N)$ 
539     a[i] *= 2; <-----  $O(1)$ 
540     for (int j = 0; j < N; j++) <-----  $O(N)$ 
541         d[i][j] = a[i] * c[j]; <-----  $O(1)$ 
542     }
543
544 for (int i = 0; i < N; i++) <-----  $O(N^2)$ 
545 { <-----  $O(i)$ 
546     a[i] *= 2; <-----  $O(1)$ 
547     for (int j = 0; j < i; j++) <-----  $O(i)$ 
548         d[i][j] = a[i] * c[j]; <-----  $O(1)$ 
549     }
550
551 for (int i = 0; i < N; i++) <-----  $O(N^2)$ 
552 {
553     if (std::find(a, a+N, 10*i) != a+N) <----- Condition:  $O(N)$  //how much time do if
statements account for?
554         count++; <-----  $O(1)$ 
555     }
556
557 for (int i = 0; i < N; i++) <-----  $O(N^2 \log N)$ 
558 {
559     a[i] *= 2; <-----  $O(1)$ 
560     for (int j = 0; j < N; j++) <-----  $O(N \log N)$ 
561         a[i][j] = f(a, N); //suppose  $f(a, N)$  is  $O(\log N)$  <-----  $O(\log N)$ 
562     }
563
564 for (int i = 0; i < R; i++) <-----  $O(Rc \log c)$ 
565 {
566     a[i] *= 2; <-----  $O(1)$ 
567     for (int j = 0; j < c; j++) <-----  $O(c \log c)$  //Insertion Sort can
be based off a constant runtime - 'order N'
568         ... f(...c...)...; //this is  $o(\log c)$ 
569     }
570
571
572 there are actually tons of different sorting algorithms - the most efficient one is
determined by the type and ordering of data
573 //compare an item to the ones before it, to figure out where it belongs. move accordingly
574 Selection Sort, Insertion Sort, Bubble Sort
575 insertion sort has a circumstance in which it's order N
576 Merge Sort - efficient, but needs additional space
577 quicksort is the fastest on average...
578
579
580 Quicksort is  $N \log N$ !
581
582 // find a pivot, move everything smaller to the left and everything larger to the right
583 // do this recursively
584
585 //Best case:  $T(N) = O(N) + 2T(N/2)$  (time to pick a pivot is  $O(N)$ , time to sort the

```

```

halves is T(N/2)
586 { _ _ _ _ _ _ _ _ }
587 [ _ _ _ _ _ ] [ _ _ _ _ _ ]
588 [ _ _ ] [ _ _ ] [ _ _ ] [ _ _ ]
589 [ ] [ ] [ ] [ ] [ ] [ ]
590
591 //Worst case: O(N) + T(N - 1) + T(0)
592 { _ _ _ _ _ _ _ _ _ _ }
593 [ _ _ _ _ _ _ _ _ _ _ ] [ _ _ ]
594
595
596 //ACTUAL
597
598 //QUICKSORT
599
600 //ALGORITHM
601 //-----
602 //swaps stuff
603 void exchange(string& x, string& y)
604 {
605     string t = x;
606     x = y;
607     y = t;
608 }
609
610 // Rearrange the elements of the array so that all the elements
611 // whose value is < separator come before all the other elements,
612 // and all the elements whose value is > separator come after all
613 // the other elements. Upon return, firstNotLess is set to the
614 // index of the first element in the rearranged array that is
615 // >= separator, or n if there is no such element, and firstGreater is
616 // set to the index of the first element that is > separator, or n
617 // if there is no such element.
618 void separate(string a[], int n, string separator,
619               int& firstNotLess, int& firstGreater)
620 {
621     if (n < 0)
622         n = 0;
623
624     firstNotLess = 0;
625     firstGreater = n;
626     int firstUnknown = 0;
627     while (firstUnknown < firstGreater)
628     {
629         if (a[firstUnknown] > separator)
630         {
631             firstGreater--;
632             exchange(a[firstUnknown], a[firstGreater]);
633         }
634         else
635         {
636             if (a[firstUnknown] < separator)
637             {
638                 exchange(a[firstNotLess], a[firstUnknown]);
639                 firstNotLess++;
640             }
641             firstUnknown++;
642         }
643     }
644 }
645
646 // Rearrange the elements of the array so that
647 // a[0] <= a[1] <= a[2] <= ... <= a[n-2] <= a[n-1]
648 // If n <= 1, do nothing.
649
650 //the meat of the function
651 void order(string a[], int n)
652 {
653     if (n <= 1)

```

```

654         return;
655     int midpoint = n / 2, firstgreater = n, firstnotless = 0;
656     separate(a, n, a[midpoint], firstnotless, firstgreater);
657
658     order(a, firstnotless);
659     order(a + firstnotless + 1, n - firstgreater);
660 }

```

```

661
662
663
664 -----
665 - At 0.25 Microseconds per step (4M Steps/sec)
666 -----

```

N	0.5 N ²	7.527 N log(2) (N)
100	1.25ms	1.25ms
1000	.125sec	1/53sec
10000	12.5sec	1/4sec
100000	21min	3.125sec
1000000	35hrs	37.5sec

```

674
675
676
677 struct Node
678 {
679     std::string data;
680     vector<Node*> children;
681 }

```

```

682
683         George
684       /  |  \
685      /   |   \
686     eliza bet  anne
687    /  \  /  |  \
688   /   \ /   |   \
689  wall  nut peg mar  owen

```

```

690
691
692 int countTree(const Node* t) //function to count the number of nodes in a tree
693 {
694     if (t == nullptr)
695         return 0;
696     int total = 1;
697     for (int k = 0; k != t->children.size(); k++)
698         total += countTree(t->children[k]);
699     return total;
700 }

```

```

701
702
703 void printTree(const Node* t) //function to print a tree
704 {
705     if (t != nullptr)
706     {
707         cout << t->data << endl;
708         for (int k = 0; k != t->children.size(); k++)
709             printTree(t->children[k]);
710     }
711 }

```

```

712
713 int fixNegatives (Node* p) //changes any negative values to 0 in a 4-tree
714 {
715     if (p == nullptr)
716         return 0;
717
718     int Count = 0;
719     if (p->data < 0)
720     {
721         p->data = 0;
722         Count++;

```

```
723     }
724     for (int i = 0; i < 4; i++)
725         Count += fixNegatives(p->child[i]);
726 }
727
728 int countIncludes(const string a1[], int n1, const string a2[], int n2) //counts the
number of times a thing is included
729 {
730     int numofInc = 0;
731     if (n1 <= 0)
732     {
733         if (n2 == 0)
734             numofInc++;
735         return numofInc;
736     }
737
738     if (a1[0] == a2[0])
739         numofInc += countIncludes(a1 + 1, n1 - 1, a2 + 1, n2 - 1);
740
741     numofInc += countIncludes(a1 + 1, n1 - 1, a2, n2);
742
743     return numofInc;
744 }
745
746
747
```