

# Assignment One – Structures & Sorting

---

Ryan Munger  
Ryan.Munger1@marist.edu

October 3, 2024

## 1 Introduction

### 1.1 Goals

Assignment 1 instructed us to create the elementary data structures *Stack* and *Queue* using a node implementation. We then checked a list of items for palindromes using these data structures and also sorted the list using selection, insertion, merge, and quick sorts.

### 1.2 Write-up Format

In this report I will describe the logic being presented. Below the text explanation, relevant code will follow. I implemented the foundational parts of this assignment in both C++ and Ada. I was originally planning to use Pascal, but became frustrated with the nuances of different versions and compilers. As I was writing Pascal, I found it incredibly similar to a language I have seen before – Ada (Not like its based on Pascal or anything...). Before you ask, yes Lockheed Martin and the defense industry is why I know Ada. I used this as an opportunity to actually learn Ada so that I can be better at my job (especially when working with simulation). I used Alire to build and run my Ada code. Alire is extremely similar to Cargo for Rust. Under the hood, it uses GNAT to build the Ada program. To compile my code (check your email), simply download GNAT studio from the AdaCore website and use GNATmake. Long story short, walrus assignment is pretty cool. I will attempt to sprinkle in some jokes for you!

### 1.3 Limerick of Luck

Limerick of Luck would make a good magic item...  
I suggest adding Nogard Dragon as a new palindrome!  
All programmers revere the power of sorting  
Especially after conducting it with no importing.  
Despite some having large big-oh,  
Merge sort hasn't yet delivered a deathblow,  
Allowing us to experiment with algorithms so cavorting.

## 2 Nodes, Linked Lists, Stacks, & Queues

### 2.1 Making a Node

Nodes are the foundations of linked lists. Other data structures including stacks and queues are special forms of linked lists. To create nodes capable of forming a singly linked list (meaning nodes know where to find the next node but not the previous), they need a value and a pointer to the following node. I used a template in my implementation so that we can use the nodes with any type of data. A nodes value and next node can be accessed publicly. The Ada implementation is limited to char nodes.

```
8 template <typename T>
9 struct Node {
10     T value;
11     Node<T>* next;
12 };

10 type Node;
11 type Node_Ptr is access Node;
12
13 type Node is
14     record
15         Data : Character;
16         Next : Node_Ptr;
17     end record;
```

### 2.2 Testing our Nodes

Code:

```
332 cout << "Node Testing:" << endl;
333 Node<string>* firstNode = new Node<string>;
334 firstNode->value = "Magic Item1";
335 Node<string>* secondNode = new Node<string>;
336 secondNode->value = "Nogard Dragon";
337 firstNode->next = secondNode;
338 cout << "First Node Value: " << firstNode->value << endl;
339 cout << "Second Node Value: " << secondNode->value << endl;
340 if (firstNode->next == secondNode) {
```

```

341     cout << "First node points to the second node." << endl;
342 } else {
343     cout << "First node does not point to the second node." <<
    endl;
344 }

```

Output:

```

1 Node Testing:
2 First Node Value: Magic Item1
3 Second Node Value: Nogard Dragon
4 First node points to the second node.

```

Code:

```

138 N1: Node_Ptr := new Node;
139 N2: Node_Ptr := new Node;
140 S: Stack;
141 Q: Queue;
142 begin
143     N1.Data := 'a';
144     N2.Data := 'b';
145     N1.Next := N2;
146     Put_Line("Node1 Value: " & N1.Data);
147     Put_Line("Node2 Value: " & N2.Data);
148     if N1.Next = N2 then
149         Put_Line("Node1 Points to Node2!");
150     else
151         Put_Line("Node1 Does not Point to Node2!");
152     end if;

```

Output:

```

1 Node1 Value: a
2 Node2 Value: b
3 Node1 Points to Node2!

```

## 2.3 Stacks & Queues

Nodes can link together to form linked lists. Stacks and queues are linked lists with special rules. Stacks can only be managed from the top, similar to a stack of dining hall plates. You can add or remove from the top of the stack but nothing more (this is called LIFO - Last In First Out). Queues are like a line to enter a dining hall; you can only add things to the end of the line and take from the front (FIFO - First In First Out).

A stack needs - push (add to stack), pop (remove), isEmpty (see if it has anything in it), and optionally, size & peek. Queues need - enqueue (add to end of line), dequeue (remove from front), isEmpty, and optionally, size & peek. All of these operations are in constant time (as my queue has a tail pointer).

The stack needs a pointer to the top of the stack just as a queue keeps track of its head and tail.

### Stack Declaration:

```
14 template <typename T>
15 class Stack {
16     private:
17         Node<T>* top;
18         int size;
19
20     public:
21         // constructor
22         Stack() {
23             top = nullptr;
24             size = 0;
25         }
26         // Destructor to free memory from nodes within stack if
27         // stack is deleted
28         ~Stack() {
29             while (!isEmpty()) {
30                 pop();
31             }
32         }
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

### Stack Push:

1. Store new data into a node.
2. The new next node is the current top.
3. The new top is the new node.
4. Increase stack size (optional).

```
37 void push(const T value) {
38     // create new node, make it the top & point to the old
39     top
40     Node<T>* newItem = new Node<T>;
41     newItem->value = value;
42     newItem->next = top;
43     top = newItem;
44     size++;
45 }
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

### Stack Pop:

1. Ensure stack not empty.
2. Note the value of current top.
3. Make the node after the current top the new top.
4. Reduce stack size (optional).
5. Delete the old node from memory (garbage collection).
6. Return the value of the old top node.

```
46     T pop() {
47         if(isEmpty()) {
48             throw std::runtime_error("Stack is empty, cannot
pop.");
49         }
50
51         T value = top->value;
52         Node<T>* newTop = top->next;
53         delete top;
54         top = newTop;
55         size--;
56
57         return value;
58     }
```

```
42     function Pop(S : in out Stack) return Character is
43         Temp_Node : Node_Ptr;
44         Value : Character;
45     begin
46         if S.Top = null then
47             raise Program_Error; -- Stack underflow?
48         else
49             Temp_Node := S.Top;
50             Value := Temp_Node.Data;
51             S.Top := S.Top.Next;
52             return Value;
53         end if;
54     end Pop;
```

### Stack isEmpty:

1. Returns true if the top of the stack is a nullptr.

### Stack getSize:

1. Returns value of size attribute.

### Stack Peek:

1. Returns the value of the top node without popping it.

---

Why did the stack go to therapy? - *It kept pushing things down...*

---

### Testing Stack:

Code:

```
349     cout << "\nStack Test: [Hello, World, !]" << endl;
350     Stack<string> stack;
351     stack.push("Hello");
352     stack.push("World");
353     stack.push("!");
354     int size = stack.getSize();
355     for (int i = 0; i < size; i++) {
356         cout << "Pop: " << stack.pop() << endl;
357     }
```

Output:

```
6 Stack Test: [Hello, World, !]
7 Pop: !
8 Pop: World
9 Pop: Hello
```

Code:

```
156     Put_Line("Stack Test: [A, B, C]");
157     Push(S, 'A');
158     Push(S, 'B');
159     Push(S, 'C');
160     for I in 1 .. 3 loop
161         Put_Line("Pop: " & Pop(S));
162     end loop;
```

Output:

```
5 Stack Test: [A, B, C]
6 Pop: C
7 Pop: B
8 Pop: A
```

As previously mentioned, the queue class will need to have a pointer to the first and the last node to work in constant time. It is possible to traverse the queue to obtain the final node, but the trade off between memory and time is worth it in this case to store an additional pointer.

### Queue Implementation:

```
70 template <typename T>
71 class Queue {
72     private:
73         Node<T>* head;
74         Node<T>* tail;
75         int size;
76
77     public:
78         Queue() {
79             head = nullptr;
80             tail = nullptr;
```

```

81         size = 0;
82     }
83
84     ~Queue() {
85         while (!isEmpty()) {
86             dequeue();
87         }
88     }

```

```

26     type Queue is
27         record
28             Front : Node_Ptr := null;
29             Rear  : Node_Ptr := null;
30         end record;

```

### Queue Enqueue:

1. Create a new node with the new value.
2. If queue empty, new node is the head.
3. Otherwise, set the current tail's next value to the new node.
4. Make the new node the tail.
5. Increment size (optional).

```

94     void enqueue(const T value) {
95         // make a new node
96         Node<T>* newItem = new Node<T>;
97         newItem->value = value;
98         newItem->next = nullptr;
99
100        // add the node to the back of the queue
101        if (isEmpty()) {
102            head = tail = newItem;
103        } else {
104            tail->next = newItem;
105            tail = newItem;
106        }
107        size++;
108    }

```

```

56     procedure Enqueue(Q : in out Queue; Value : Character) is
57         New_Node : Node_Ptr := new Node;
58     begin
59         New_Node.Data := Value;
60         New_Node.Next := null;
61
62         if Q.Rear = null then
63             -- Queue is empty, so Front and Rear point to the new node
64             Q.Front := New_Node;
65             Q.Rear := New_Node;
66         else
67             -- Append to the rear of the queue
68             Q.Rear.Next := New_Node;

```

```

69         Q.Rear := New_Node;
70     end if;
71 end Enqueue;

```

### Queue Dequeue:

1. Make sure queue is not empty.
2. Note the value of the current head.
3. Set the new head to the current head's next.
4. Free up memory from the now old head.
5. Check if the head is now null, if so, set tail to null.
6. Decrement size (optional).
7. Return the value of now the old head node.

```

110     T dequeue() {
111         if(isEmpty()) {
112             throw std::runtime_error("Queue is empty, cannot
113 dequeue.");
114         }
115
116         T value = head->value;
117         Node<T>* oldHead = head;
118         head = oldHead->next;
119         delete oldHead;
120         size--;
121
122         if (head == nullptr) {
123             tail = nullptr;
124         }
125         return value;
126     }

```

```

73 function Dequeue(Q : in out Queue) return Character is
74     Temp_Node : Node_Ptr;
75     Value : Character;
76 begin
77     if Q.Front = null then
78         raise Program_Error; -- Queue underflow?
79     else
80         Temp_Node := Q.Front;
81         Value := Temp_Node.Data;
82         Q.Front := Q.Front.Next;
83
84         -- if front null its empty so rear null
85         if Q.Front = null then
86             Q.Rear := null;
87         end if;
88
89         return Value;
90     end if;
91 end Dequeue;

```



#### Queue isEmpty:

1. Return true if the head and the tail are null.

#### Queue peekFront:

1. Return value of head node.

#### Queue peekBack:

1. Return value of tail node.

#### Queue getSize:

1. Return value of size attribute.

---

Why did the stack break up with the queue? - *It found a priority queue to put it first...*

---

#### Testing Queue:

Code:

```
359     cout << "\nQueue Test: [Hello, World, !]" << endl;
360     Queue<string> q;
361     q.enqueue("Hello");
362     q.enqueue("World");
363     q.enqueue("!");
364     size = q.getSize();
365     for (int i = 0; i < size; i++) {
366         cout << "Dequeue: " << q.dequeue() << endl;
367     }
```

Output:

```
11 Queue Test: [Hello, World, !]
12 Dequeue: Hello
13 Dequeue: World
14 Dequeue: !
```

Code:

```
166     Put_Line("Queue Test: [A, B, C]");
167     Enqueue(Q, 'A');
168     Enqueue(Q, 'B');
169     Enqueue(Q, 'C');
170     for I in 1 .. 3 loop
171         Put_Line("Dequeue: " & Dequeue(Q));
172     end loop;
```

Output:

```
10 Queue Test: [A, B, C]
11 Dequeue: A
12 Dequeue: B
13 Dequeue: C
```

## 2.4 Palindrome Checking: A Stack & Queue Use Case

A palindrome is word/phrase that reads the same forward and backwards. For our implementation, we will ignore spaces and case. "Racecar" is a palindrome because the letters read the same front to back and back to front. "Nogard Dragon" should be in the ones we are checking because I think its cool (is that not reason enough?). We read in the list of words/phrases to check from magicitems.txt and determine if it is a palindrome by:

1. Declaring a stack and a queue.
2. For each char in the item, push/enqueue each char to each respective data structure in lower case if it is not a space.
3. Pop each element from the stack and dequeue each element from the queue. If they do not match, it is not a palindrome! (As stack has the word in reverse order).

Read in magic items:

```
148 vector<string> getMagicItems(const string& filename) {
149     vector<string> magicItems;
150     ifstream file(filename); //input file stream
151     if (!file) {
152         cerr << "File opening failed." << endl;
153     }
154     string line;
155     while (getline(file, line)) {
156         magicItems.push_back(line); // add line to vector
157     }
158     file.close();
159     return magicItems;
160 };

120 procedure Check_Magicitems(File_Name : String) is
121     File : File_Type;
122     Line : String (1 .. 100); -- assuming max line length is 100
123     characters
124     Length : Natural;
125     Count : Integer := 0;
126     begin
127         Open (File, In_File, File_Name);
128         while not End_Of_File (File) loop
129             Get_Line (File, Line, Length);
130             if Is_Palindrome(Line (1 .. Length)) then
131                 Put_Line(""" & Line (1 .. Length) & "" is a
132                 palindrome."); -- double "" is the escape sequence...
133                 Count := Count + 1;
134             end if;
135         end loop;
136         Close (File);
137         Put_Line("Total Palindromes Found: " & Integer'Image(Count));
138         -- interesting type cast here
139     end Check_Magicitems;
```

Check for palindromes:

```
162 bool isPalindrome(const string& word) {
163     Stack<char> s; Queue<char> q;
164     for (char letter: word) {
165         if (!isspace(letter)) {
166             if (isalpha(letter)) {
167                 letter = tolower(letter);
168             }
169             s.push(letter);
170             q.enqueue(letter);
171         }
172     }
173
174     int size = s.getSize();
175     for (int i = 0; i < size; i++) {
176         if (s.pop() != q.dequeue()) {
177             return false;
178         }
179     }
180     return true;
181 };
```

```
93 function Is_Palindrome(Input : String) return Boolean is
94     S : Stack;
95     Q : Queue;
96     Len : Integer := Input'Length;
97 begin
98     -- Load the string into both the stack and the queue
99     -- Ignore case and space
100     for I in 1 .. Len loop
101         if Input(I) /= ' ' then
102             Push(S, Ada.Characters.Handling.To_Lower(Input(I)));
103             Enqueue(Q, Ada.Characters.Handling.To_Lower(Input(I)));
104         end if;
105     end loop;
106
107     -- Pop, dequeue, compare
108     for I in 1 .. Len loop
109         if Input(I) /= ' ' then
110             if Pop(S) /= Dequeue(Q) then
111                 return False;
112             end if;
113         end if;
114     end loop;
115
116     return True;
117 end Is_Palindrome;
```

Results:

```
16 Palindrome Checking:
17 "Boccob" is a palindrome!
18 "Seuss Igniting Issues" is a palindrome!
19 "UFO tofu" is a palindrome!
20 "Ebuc Cube" is a palindrome!
21 "Aibohphobia" is a palindrome!
22 "Taco cat" is a palindrome!
```

```

23 "Was It A Rat I Saw" is a palindrome!
24 "Olah Halo" is a palindrome!
25 "CD case divides ACDC" is a palindrome!
26 "Dacad" is a palindrome!
27 "Dior Droid" is a palindrome!
28 "Robot Tobor" is a palindrome!
29 "Narc in a panic ran" is a palindrome!
30 "radar" is a palindrome!
31 "Golf flog" is a palindrome!
32 Total Palindromes Found: 15

```

---

Why is 'palindrome' not a palindrome? Dr. Labouseur and other detail oriented people surely don't get annoyed by this...

---

## 3 Sorting

### 3.1 Introduction

In this section we will sort our list alphabetically from least (a) to greatest (z). The  $\leq$  relation is a **total order** and must uphold the following:

1. Reflexivity: For all  $x$  in  $S$ ,  $x \leq x$ .
2. Antisymmetry: For all  $x, y$  in  $S$ , if  $x \leq y$  and  $y \leq x$ , then  $x = y$ .
3. Transitivity: For all  $x, y, z$  in  $S$ , if  $x \leq y$  and  $y \leq z$ , then  $x \leq z$ .

Good 'ol discrete math...

I implemented this section in c++ only.

### 3.2 Unsorting?

A truly unsorted array is one in which the elements are randomly positioned. This array can technically be sorted (the basis for the monkey/bogo sort), but we won't actually know until we check. Since we are implementing several sorting algorithms, we will need to unsort/shuffle the array between sorts. A good algorithm to do this randomly & in place is the Knuth/Fisher Yates Shuffle. It iterates through each data point in an array, picking a random index between current index and all remaining indices and swaps the values between the two.

```

140 void knuthShuffle(vector<string>& arr) {
141     int n = arr.size();
142     for (int i = n-1; i > 0; i--) {
143         int random = rand() % (i + 1); // inclusive to swap in
144         place swap(arr[i], arr[random]);
145     }
146 };

```

### 3.3 Selection Sort

Selection sort is an in place sort that organizes an array using a nested loop. Selection sort iterates through the array except for the portion it has already sorted and finds the minimum value (when doing  $\leq$ ). Once we find the minimal element, we swap it with the current index. This way, if we are on array index 2 (starting at 0) we would swap in the 3rd smallest value. This algorithm works in  $O(n^2)$  time because the outer loop runs  $n$  times and the inner loop has to iterate over the entire array it hasn't yet sorted. Despite the fact that the inner loop decreases in iterations as the sort progresses, this is still quadratic time because the number of the inner iterations depends on the size of the input; therefore  $n*n$  comparisons (where  $n$  is size of the array). A more precise characterization of this algorithm is as follows:

Total comparisons:  $(n-1) + (n-2) + \dots + 1 = \sum_{i=1}^{n-1} i$

$$\sum_{i=1}^{n-1} i = \frac{(n-1)+1}{2}(n-1) = \frac{1}{2}n(n-1) = \frac{1}{2}(n^2 - n)$$

As we can see, in this case  $n^2$  dominates therefore the algorithm is  $O(n^2)$ . Using this formula on our array of 666 elements, we can expect  $\frac{1}{2}(666^2 - 666)$  or 221,445 comparisons to be made. It will always take this exact number of comparisons, even if the array is presorted or totally unsorted.

```
192 int selectionSort(vector<string>& arr) {
193     int n = arr.size();
194     int comparisons = 0;
195
196     for (int i = 0; i < n; i++){
197         int minIdx = i;
198         for (int j = i+1; j < n; j++) {
199             comparisons++;
200             if (toLowerCase(arr[j]) < toLowerCase(arr[minIdx])){
201                 minIdx = j;
202             }
203         }
204         if (minIdx != i) {
205             swap(arr[i], arr[minIdx]);
206         }
207     }
208     return comparisons;
209 }
```

---

Why did selection sort's girlfriend leave him? - *Turns out checking out every option before selecting her wasn't flattering...*

---

### 3.4 Insertion Sort

Insertion sort is an in place sort that is similar to selection sort. Instead of iterating through the entire array to find the item destined for the next position in the array, insertion sort simply grabs the next item and puts it in the correct location. Instead of searching forward for the most optimal value, insertion sort *inserts* the next value it is looking at in its correct relative position. The exact time complexity of this algorithm is difficult to calculate directly as it

depends on the order the array is in, but the worst case is that the inner loop will have to iterate through the entire sorted portion of the array to find the correct location for each item. The worst case makes insertion sort  $O(n^2)$  since the length of the inner array is approximately  $n/2$  on average. A complexity of  $n * \frac{1}{2}n$  without constants is  $n^2$ . For our array of 666 elements, we can expect between best case (sorted:  $n - 1$  comparisons) or worst case (same as selection sort), or 665 - 221,445. Averaging these, we should expect generally around 111,388 comparisons.

```

212 int insertionSort(vector<string>& arr) {
213     int n = arr.size();
214     int comparisons = 0;
215
216     for (int i = 1; i < n; i++) {
217         int insertIdx = i;
218         string currentCheck = arr[i];
219         for (int j = i-1; j >= 0; j--) {
220             comparisons++;
221             if (toLowerCase(arr[j]) > toLowerCase(currentCheck)) {
222                 arr[j+1] = arr[j];
223                 insertIdx = j;
224             } else {
225                 break;
226             }
227         }
228         arr[insertIdx] = currentCheck;
229     }
230     return comparisons;
231 };

```

### 3.5 Merge Sort

Merge sort is a divide and conquer algorithm that is not in place. It works by recursively dividing the array into smaller subarrays and sorting those subarrays. The subarrays are sorted by dividing them just as we did with the input array. The base case is an array of size one, which is sorted! Once we reach subarrays of size one, we then merge them back together all the way up the recursive tree until we have a completely sorted array.

To be concise, merge sort divides the array into halves, sorts each half, and then merges the sorted halves back together (and does so for each half of the half... etc.). This process is repeated until the entire array is sorted. Dividing the array is a constant time operation. There are  $\log_2 n$  divides as the array is cut in half each recursion. Each divide step must also be merged back together. There are  $n$  items to merge back together at each subarray size. This operation will take  $n$  time. Therefore, the time complexity is  $O(n \log_2 n)$ . We should then expect  $666 * \log_2 666$  which is 6,244 comparisons.

Merge sort is driven by a recursively called function that

1. Checks if the subarray can be divided further. If not, array is of size 1 and is sorted (base case).

2. Makes a recursive call to divide the subarray further if it can.
3. Calls a helper function to stitch the two halves back together in a sorted order.

```

280 void mergeSort(vector<string>& arr, int left, int right, int&
    comparisons) {
281     // base case
282     if (left >= right)
283         return;
284
285     // left < right, array can be divided further
286     // calculate mid of subarray/array to split array
287     int mid = left + (right - left) / 2;
288
289     // recursively sort the halves, update comparisons
290     mergeSort(arr, left, mid, comparisons);
291     mergeSort(arr, mid + 1, right, comparisons);
292
293     // merge sorted halves, update comparisons
294     merge(arr, left, mid, right, comparisons);
295 };
```

The helper function first builds the two subarrays we are to work with (the only subarrays that get to this function are sorted). We then write the subarray data back into the original array in the correct order. Combining the two arrays is simple as we can just use a while loop to see which next element in either array is smaller since the subarrays are sorted. Once this is completed, the two subarrays have been merged successfully.

```

233 // referenced geeksforgeeks.org
234 void merge(vector<string>& arr, int left, int mid, int right, int&
    comparisons) {
235     int nLeft = mid - left + 1; // size of left array
236     int nRight = right - mid; // size of right array
237
238     // dec temp arrays
239     vector<string> leftArr(nLeft), rightArr(nRight);
240
241     // populate temp vectors
242     for (int i = 0; i < nLeft; i++)
243         leftArr[i] = arr[left + i];
244     for (int j = 0; j < nRight; j++)
245         rightArr[j] = arr[mid + 1 + j];
246
247     int i = 0, j = 0;
248     int k = left;
249
250     // merge temp vectors back into arr[left..right]
251     while (i < nLeft && j < nRight) {
252         comparisons++;
253         if (toLowerCase(leftArr[i]) <= toLowerCase(rightArr[j])) {
254             arr[k] = leftArr[i];
255             i++;
256         }
257         else {
```

```

258         arr[k] = rightArr[j];
259         j++;
260     }
261     k++;
262 }
263
264 // copy the remaining elements of leftArr after merging
265 while (i < nLeft) {
266     arr[k] = leftArr[i];
267     i++;
268     k++;
269 }
270
271 // copy the remaining elements of right after merging
272 while (j < nRight) {
273     arr[k] = rightArr[j];
274     j++;
275     k++;
276 }
277 };

```

---

A mad scientest has created a self sorting array! He pulls back the curtain and to the crowd's dismay yells: "*BEHOLD! A array of size one!!*"

---

### 3.6 Quick Sort

Quick sort also utilizes a divide and conquer methodology. Similar to merge sort, it recursively divides the array using a pivot value. It partitions the array by splitting the array at the pivot value by grouping items less than the pivot into one partition and the rest into another. There are many different versions of the partition algorithm, but I chose the Lomuto partition as it is easy to understand. Lomuto partition keeps track of the indices of smaller elements and keep swapping, always using the last element as the pivot. The image below from [GeeksforGeeks.org](https://www.geeksforgeeks.org/quick-sort/) does a great job of visualizing the process.

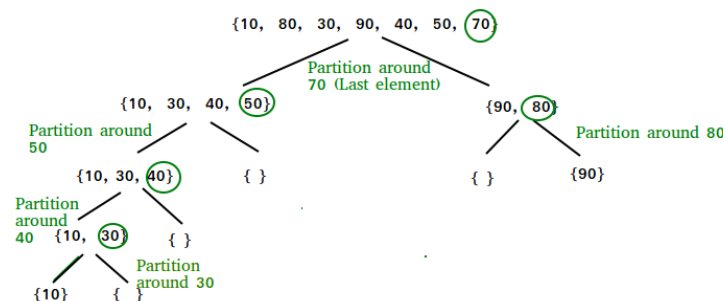


Figure 1: Lomuto Quick Sort



The time complexity for quick sort is  $O(n \log_2 n)$  as there are rarely more than  $\log_2 n$  partitions (given we are choosing good pivots) and  $n$  comparisons required for each partition to correctly order the elements around the pivot. The absolute worst case is that the pivot is always the largest or smallest element, which leaves us at  $O(n^2)$ , or 221,445 comparisons. On average however, we can expect  $O(n \log_2 n)$ , or  $666 * \log_2 666$  which is 6,244 comparisons.

First, we have the recursive function:

1. Checks if the partition can be divided further. If not, array is of size 1 and is sorted (base case).
2. Call a helper function to partition the array.
3. Makes a recursive call to sort the partitions.

```

320 void quickSort(vector<string>& arr, int low, int high, int&
    comparisons) {
321     // if low < high then we are not done
322     if (low < high) {
323         int pIdx = partition(arr, low, high, comparisons); //
    Partitioning index
324
325         quickSort(arr, low, pIdx - 1, comparisons); // recursively
    sort the elements before partition
326         quickSort(arr, pIdx + 1, high, comparisons); // recursively
    sort the elements after partition
327     }
328 };

```

The helper function is a bit more complex (we are using Lomuto).

1. Choose a pivot value (last element in partition).
2. Locate the start index of the partition.
3. Traverse partition and move all smaller elements to the left side of the pivot. This generally leaves the pivot toward the middle of the array. We can then again partition further.

Once again, GeeksforGeeks.org has an excellent graphic:

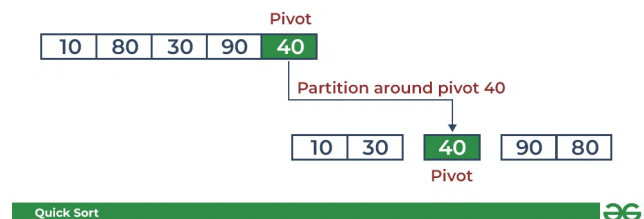


Figure 2: Pivoting

```

296 // referenced geeksforgeeks.org
297 int partition(vector<string>& arr, int low, int high, int&
    comparisons) {
298     // choose pivot element
299     string pivot = toLowerCase(arr[high]);
300     // Index of smaller element + right position of pivot found so
    far
301     int i = low - 1;
302
303     // rearranges array so that elements less than or equal to the
    pivot are to the left of it
    // greater than are to the right
304     for (int j = low; j <= high - 1; j++) {
305         comparisons++;
306         if (toLowerCase(arr[j]) < pivot) {
307             i++;
308             swap(arr[i], arr[j]);
309         }
310     }
311
312     // move pivot after smaller elements
313     swap(arr[i + 1], arr[high]);
314     return i + 1;
315 }
316 };

```

Why does quicksort have no friends? - *It always pivots the conversation to talk about itself...*

---

### 3.7 Comparing the Sorts

In terms of time complexity, quick sort and merge sort easily pull ahead of selection and insertion. This does not mean that selection and insertion sorts are not useful. Selection sort is an excellent teaching exercise, is easy to understand, and employs minimal overhead & swaps. Insertion sort is very quick to sort already sorted/near sorted arrays. Merge sort generally uses less comparisons than quick sort, but ends up with more overhead. Most in-built sorting libraries like `sort()` in Python use Timsort. Timsort is simply a hybrid sorting algorithm derived from ((merge sort or quick sort) and insertion sort). By using both algorithms, we can play on the strengths of each. By using merge sort to get the array almost sorted and insertion to finish the job, improvements are made. Since the code I implemented counts the comparisons for each sort, I will run it 20 times and average the results in a table below.

Sort	Time Complexity	Expected	Actual
Selection Sort	$O(n^2)$	221,445	221,445
Insertion Sort	$O(n^2)$	111,388	112,609
Merge Sort	$O(n \log_2 n)$	6,244	5,424
Quick Sort	$O(n \log_2 n)$	6,244	6,760

Selection sort is exactly as expected since it always checks every element. Insertion sort did vary a bit, but is right on par with the expected. Merge sort

performed well, as it is usually a bit faster than its Big-Oh. Quick sort lagged behind, but this is likely due to my suboptimal choice of the Lomuto partitioning algorithm.

## 4 Miscellaneous Lessons Learned

If I learned anything, it is that the walrus operator is sick. I guess I learned a lot about Big Oh and sorting algorithms, but that pales in comparison. One thing I wished I knew is that you need to seed the random number generator in C++ or it will always give you the same numbers (seeding with current time is best). Aside from general Ada knowledge (why are we not using \ to escape things??), I am satisfied with my newfound understanding of fundamental data structures and sorting algorithms.

---

You really read this whole thing? **WOW!**

---