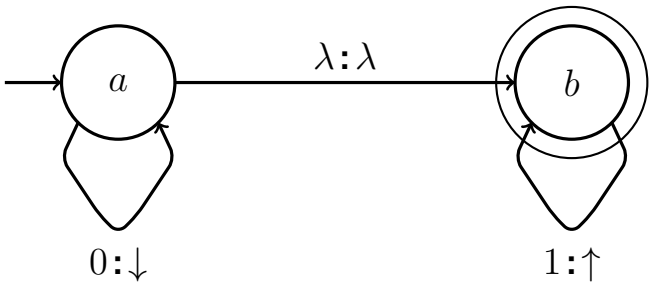
Programming Dictionary

By Charles Cook

Begun on Dec. 6th, 2018

1 Compiler Theoretical Foundations

Push Down Automata for strings of the form 0^n1^n



Key:

Symbol	Meaning
a, b	State Node
0, 1	Symbol to Print
$\downarrow$	Push (onto the stack)
$\uparrow$	Pull (off of the stack)
λ	Null operation (no print, push, or pull)

2 Programming Tools & Languages: First Ten Questions

2.1 Swap Function

2.1.1 Pseudocode for *swap*

```
void swap(a, b) {
        temp = a;
        a = b;
        b = temp;
}

void swapNoTemp(a, b) {
        a += b; // a = a + b
        b -= a; // b = b - (a + b) = -a
        b *= -1; // b = a
        a -= b; // a = a + b - a = b
}
```

2.1.2 *swaptest.c*

```
#include <stdio.h>
void swap(int *a, int *b) {
        *a += *b;
        *b -= *a;
        *b *= -1;
        *a -= *b;
}

int main() {
        int x, y, z;
        x = 10;
        y = 13;
        z = 2;

        swap(&x, &y);
        swap(&x, &z);
        swap(&y, &z);
        printf("x: &d\ny: &d\nz: &d\n", x, y, z);

        return 0;
}
```

2.1.3 *swaptest.cpp*

```
#include <iostream>
using namespace std;

void swap(int *a, int *b) {
        *a += *b;
        *b -= *a;
        *b *= -1;
        *a -= *b;
}

int main() {
        int x, y;
        x = 13;
        y = 29;

        cout << x << ", " << y << "\n";
        swap(x, y);
        cout << x << ", " << y << "\n";

        return 0;
}
```

2.2 Reverse an array with no extra space (pseudocode)

```
void reverseArray(array, int length) {
        for (int i = 0; i < length / 2; i++) {
                swap(array[i], array[length - i - 1]);
        }
}
```

2.3 Reverse a doubly linked list (pseudocode)

```
struct Node {
        int data;
        struct Node *next;
        struct Node *prev;
};

void reverseDLL(head, tail) {
        struct Node tempH = head;
        struct Node tempT = tail;

        while (tempH -> next != tempT -> prev && tempH != tempT) {
                swap(tempH -> data, tempT -> data);
        }
}
```

2.4 Reverse a doubly linked list recursively (pseudocode)

```
void reverseDLL_Recursive(head, tail) {
        if (head -> next != tail -> prev && head != tail) {
                swap(head -> data, tail -> data);
                reverseDLL_Recursive(head -> next, tail -> prev);
        }
}
```

2.5 Pointer to a Pointer

A use of a pointer-to-a-pointer in C would be to insert an element into a sorted singly-linked list. Take the following list:

```
head:
 [2] -> [4] -> [7] -> [10] -> [13]
```

Say we want to insert the node **[9]** into the list; If we make a pointer-to-pointer, call it **pp**, and make it point to the **next** attribute of each node, we can insert **[9]** as follows:

First, we set the value of **pp** to be the **head** of the list:

***pp = head;**

where head is the pointer to the first struct in the list. We can visualize this below:

```
 pp
  |
  V
head:
 [2] -> [4] -> [7] -> [10] -> [13]
```

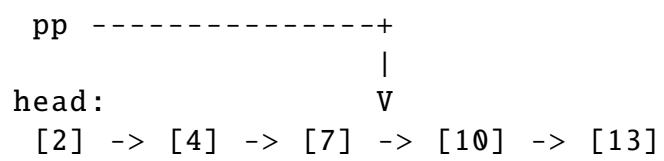
***pp** is **[2]**, and **[2] -> data** is 2, which is less than 9. Until we find a candidate ***pp -> data** that is greater than 9, we keep going by setting **pp** to the address of the next node with ***pp = &(*pp -> next)**.

```
 pp -+
     |
head:V
 [2] -> [4] -> [7] -> [10] -> [13]
```

***pp** is **[2] -> next** which is **[4]**, **[4] -> data** is 4 which is less than 9, keep going; ***pp = &(*pp -> next)**

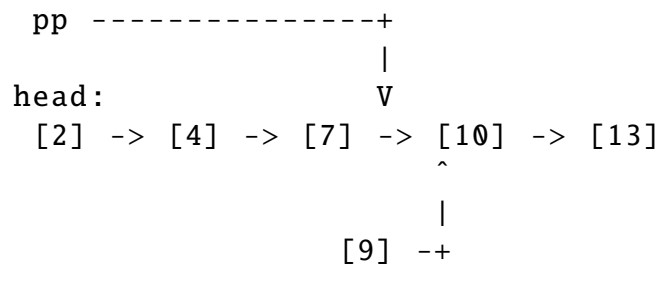
```
 pp --------+
            |
head:       V
 [2] -> [4] -> [7] -> [10] -> [13]
```

***pp** is **[4] -> next** which is **[7]**, **[7] -> data** is 7 which is less than 9, keep going; ***pp = &(*pp -> next)**

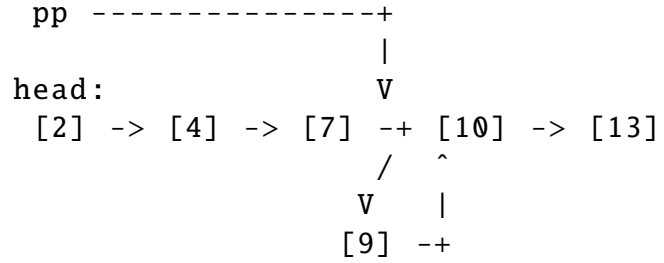


Paydirt! ***pp -> data** is 10, which is greater than 9. Now we do the following:

[9] -> next = *pp;



***pp = [9];**



The list is still ordered, and **pp** is needed no longer.

2.6 Sorting Algorithm with $O(n)$ Complexity

An optimised Bubble Sort that stops after doing a pass thru an array wherein no swaps occur will run in $O(n)$ time on an already-sorted list. In a sorted list, comparing every element with its next element all the way to the end will not prompt Bubble Sort to perform any swaps at all. When a single pass of the array is completed and no swaps have occured, a proper Bubble Sort will stop, as no swaps is a sign of a sorted list.

2.7 Selection Sort Complexity

Consider the worst case of Selection Sort: a completely-reversed array, say $(6, 5, 4, 3, 2, 1)$. On the first pass, 1 is selected, and has to be moved 5 indices, $n - 1$. On the second pass, 2 is selected, and has to be moved 4 indices, $n - 2$. This eventually boils down to $n - 1 + n - 2 + n - 3 + n - 4 + n - 5$ operations; given $n = 6$, this becomes $6 - 1 + 6 - 2 + 6 - 3 + 6 - 4 + 6 - 5 = 5 + 4 + 3 + 2 + 1 = 5 * 6/2$ (per Gauss' Formula). If we reintroduce n, we get $\frac{(n-1)*n}{2} = \frac{n^2-n}{2}$. Overall, Selection Sort's worst case (and average case) boils down to $O(n^2)$.

2.8 Insertion Sort Complexity

Consider again $(6, 5, 4, 3, 2, 1)$. On the first pass of Insertion Sort, the 5 is swapped 1 time, $n - 5$. On the second pass, the 4 is swapped 2 times, $n - 4$. Note the pattern. Overall, there are $n-5+n-4+n-3+n-2+n-1$. Recall from the previous problem on Selection Sort that this boils down to $\frac{(n-1)n}{2} = \frac{n^2-n}{2}$. Insertion Sort is also worst-case and average-case $O(n^2)$ complexity.

2.9 *ordered* Predicate in Prolog

2.9.1 *sorted.pl*

```
ordered([X]).
ordered([X, Y|Z]) :- X < Y, ordered([Y|Z]).
```

2.9.2 Output on $[1, 2, 4, 8, 16]$ with trace-on in GProlog

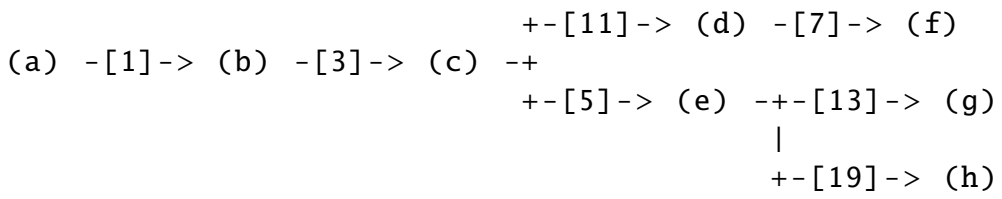
```
| ?- ordered([1,2,4,8,16]).
      1    1  Call: ordered([1,2,4,8,16]) ?
      2    2  Call: 1<2 ?
      2    2  Exit: 1<2 ?
      3    2  Call: ordered([2,4,8,16]) ?
      4    3  Call: 2<4 ?
      4    3  Exit: 2<4 ?
      5    3  Call: ordered([4,8,16]) ?
      6    4  Call: 4<8 ?
      6    4  Exit: 4<8 ?
      7    4  Call: ordered([8,16]) ?
      8    5  Call: 8<16 ?
      8    5  Exit: 8<16 ?
      9    5  Call: ordered([16]) ?
      9    5  Exit: ordered([16]) ?
      7    4  Exit: ordered([8,16]) ?
      5    3  Exit: ordered([4,8,16]) ?
      3    2  Exit: ordered([2,4,8,16]) ?
      1    1  Exit: ordered([1,2,4,8,16]) ?

true ?

yes
{trace}
| ?-
```

2.10 Labelled Path (with cost) in Prolog

2.10.1 ASCII-art chart of my path



2.10.2 *charliesPath.pl*

```
link(a, b, 1).
link(b, c, 3).
link(c, d, 11).
link(c, e, 5).
link(d, f, 7).
link(e, g, 13).
link(e, h, 19).

path(Origin, Destination, Distance) :- link(Origin, Destination, Distance).
path(Origin, Destination, TotalDistance) :-
        link(Origin, Intermediate, DistanceA),
        path(Intermediate, Destination, DistanceB),
        TotalDistance is DistanceA + DistanceB
.
```

2.10.3 Recursion Analysis

Whenever path is called with two locations that are indirectly linked, the first thing that will happen is that all directly linked locations to the Origin will be checked; these are the Intermediate locations. These locations will eached be checked to see if they have a direct link to the Destination, otherwise their directly linked locations will act as new Intermediates and themselves go through the same check.

Once an Intermediate is found that directly connects to the Destination, the recursion will cease and the value of true will be returned, along with the total distance. If all recursion becomes exhausted and no satisfactory Intermediate is found, the value of false will be returned.