Contents

Note

Everything is 1 indexed, despite using vaguely Pythonic syntax. This means $A[\text{len}(A)] = A[-1]$. Slicing is $A[a:b] = [A_a, A_{a+1}, \ldots, A_{b-1}]$. Where bounds checking is obviously necessary it is omitted. I assume a different memory model from Python: each entry of $B = [[]]$ is an independent list.

Part 1. **Foundations**

1. Insertion Sort

Maintains the invariant that $A[1:j-1]$ is sorted by shifting elements right. Insertion sort is *stable*, i.e. two keys already in sorted order remain in the same order at the end. Running time is $O\left(n^2\right)$.

```
Insertion-Sort(A)
1  for j = 2 to len(A):
2      key = A[j]
3      i = j − 1
4      while i > 0 and A[i] > key:
5          A[i + 1] = A[i]
6          i = i − 1
7      # either we're one passed the left end
8      # or A[i] ≤ key and so
9      # A[i + 1] is the proper place for key
10     A[i + 1] = key
```

2. Selection Sort

Maintains the same invariant as Insertion Sort but does so by going forward and *selecting* the smallest element each time. Running time is $O\left(n^2\right)$.

```
Selection-Sort(A)
1  for j = 1 to len(A):
2      A[j] = min(A[j + 1 :])
```

3. Bubble Sort

"Bubble up" pair by pair. Stop when no more "bubblings" are possible. Running time is $O\left(n^2\right)$.

```
Bubble-Sort(A)
1  flips = True
2  while flips:
3      flips = False
4      for i = 1 to len(A) − 1:
5          if A[i] > A[i + 1]:
6              A[i], A[i + 1] = A[i + 1], A[i]
7              flips = True
```

4. Merge Sort

Divide and conquer approach. Divide the array in half, recurse, combine results by merging, i.e. taking the smallest entry from each piece in turn. Base case is just an array with one element. Running time is $O(n \lg n)$

```
Merge-Sort(A)
if len(A) == 1:
    return A
else:
    h = ⌊len(A)/2⌋
    L = Merge-Sort(A[h:])
    R = Merge-Sort(A[1:h])
    M = [ ]
    while len(L) > 0 and len(R) > 0:
        # take the minimum of the {L[1], R[1]}
        # and remove it from further contention
        if L[1] < R[1]:
            M.append(L[1])
            del L[1]
        else:
            M.append(R[1])
            del R[1]
    # one of L, R is large by one element.
    if len(L) > 0
        M.append(L[1])
    else:
        M.append(R[1])
        M.append(R[-1])
    return M
```

5. Binary search

If an array is already sorted then you can find an element in it faster than $O(n)$ time; you can find it in $O(\lg n)$ time. Search in either the left side of the middle entry or the right side.

```
Binary-Search(A, x)
1 if x == A[h]:
2     return True
3 elif x < A[h]:
4     return Binary-Search(A[1 : h])
5 else:
6     return Binary-Search(A[h :])
```

6. Horner's Rule

Given $A = [a_1, \ldots, a_n]$ the coefficients of a polynomial and a value x a faster way to calculate $p(x)$ is

$$p(x) = \sum_{k=1}^{n} a_k x^k = a_1 + x\left(a_2 + x\left(a_3 + \cdots + x\left(a_{n-1} + xa_n\right)\right)\right)$$

```
Horners-Rule(A, x)
1 y = 0
2 for i = n downto 1:
3     y = A[i] + x · y
```

7. Maximum Positive Subarray/Kidane's algorithm

Given $A = [a_1, \ldots, a_n]$, how to find the subarray with the maximum positive sum? Use dynamic programming solution Kidane's algorithm. Change the problem to look at maximum sum subarray ending at some j. Maximum sum subarray ending at j is either empty, i.e. has negative sum, in which case its sum is 0, or includes $A[j]$. The maximum sum subarray in all of A is the maximum of all subarrays ending at all j. Running time is $\Theta(n)$.

```
Kidane-Max-Subarray(A)
1 mHere = mAll = A[1]
2 for i = 2 to len(A):
3     mHere = max(0, mHere + A[i])
4     mAll = max(mAll, mHere)
5 return mAll
```

Note that if at $j-1$ the subarray was empty, and hence $maxHere = 0$ then at j it's the case that $maxHere = A[j]$. In order to recover the actual subarray you need to keep track of whether counting is reset or subarray is extended. Easiest way to do this is using Python tricks.

```
Kidane-Max-Subarray-Mod(A)
mHere = mAll = [[ ], A[1]]
for i = 2 to len(A):
    # take max wrt. first entry of arguments, i.e. max(0, mHere + A[i])
    mHere = max([0, [ ]], [mHere + A[i], mHere.append(A[i])], key=itemgetter(1))
    mAll = max(mAll, maxHere, key=itemgetter(1))
return mAll
```

8. Reservoir Sampling

8.1. **Unweighted simple.** Suppose you want to sample k items from n items $A = [a_1, \ldots, a_n]$ fairly, i.e. uniform random, **without replacement**, draws. If you have all n items available immediately then this is simple, but if you're solving the problem *online* it's slightly more involved. For example you might not want to store all n items. Put the first k items into a *reservoir* R then for item $i > k$ draw $j \in \{1, \ldots, i\}$ inclusive. If $i \leq k$ the replace ith item. Running time is $\Theta(n)$.

```
Unweighted-Reservoir-One(A, k)
R = [a_0, a_1, ..., a_k]
for i = k + 1 to len(A):
    j = Random(1, i) # both ends inclusive
    if j ≤ k:
        R[j] = A[i]
```

8.2. **Unweighted slightly more involved.** Another way to do solve the same problem is to use a priority queue. Why complicate things? This solution generalizes to weighted sampling. Running time takes $O(n \lg k)$ because of potentially n `Extract-Min` operations on a k length priority queue.

```
Unweighted-Reservoir-Two(A, k)
R = Min-Priority-Queue
for i = 1 to k:
    u ~ Uniform(0, 1)
    # priority key is first entry in argument
    H.insert(u, A[i])
for i = k + 1 to len(A):
    u ~ Uniform(0, 1)
    # H.min returns value of minimum without extracting
    if u < H.min:
        H.Extract-Min()
        H.insert(u, A[i])
```

8.3. **Weighted.** Suppose the same sampling problem but each element has a weight associated with it. `Unweighted-Reservoir-Two` extends naturally (sort of).

```
Weighted-Reservoir(A, k)
1  R = Min-Priority-Queue
2  for i = 1 to k:
3      u ~ Uniform(0, 1)
4      u = u^(1/A[i].weight)
5      H.insert(u, A[i])
6  for i = k + 1 to len(A):
7      u ~ Uniform(0, 1)
8      u = u^(1/A[i].weight)
9      if u < H.min:
10         H.Extract-Min()
11         H.insert(u, A[i])
```

9. Online Maximum

Suppose you wanted to compute a maximum of n items but we can only make the selection once. This is similar to online sampling: fill a reservoir R full of candidates and pick the maximum from the reservoir. Then after finding that maximum pick the next maximum (if one exists) that's higher; this will be the single selection. But what size should the reservoir be? Turns out if $k = n/e$ where e is $\exp(1)$ then we'll pick the true maximum with probability at least e^{-1}. This can be further simplified by realizing you don't need to keep the entire reservoir and you can return after the first forthcoming maximum (if one exists).

```
Online-Max(A, n)
1  m = A[1]
2  # these selections to count against the quota
3  for i = 2 to ⌈n/e⌉:
4      if A[i] > m:
5          m = A[i]
6  # this one is for keeps
7  for i = k + 1 to len(A):
8      if A[i] > m:
9          return A[i]
```

10. Stable Matching

The task is given n men and n women, where each person has ranked all members of the opposite sex in order of preference, marry the men and women together such that there are no two people of

opposite sex who would both rather have each other than their current partners (a stable matching). One question is does such a stable matching even exist? In fact it does and the algorithm that produces one, the Gale-Shapley algorithm, proves it. It runs in $O\left(n^2\right)$. The next question is the solution optimal. In fact it is not. The algorith is simple: first each man proposes to the woman he prefers best and each woman accepts provisionally, i.e. accepts a proposal but trades up if a more desirable man proposes. Do this for n rounds (or until there are no more unengaged men). Running time is $O\left(n^2\right)$

Matching(P_m, P_w, men)

```
1  # men is an array of men to be matched
2  # P_m is an n × n preferences matrix for the men, sorted by increasing priority
3  # P_w is an n × n a preferences matrix for the women, sorted
4  matched_M = {}
5  matched_W = {}
6  while len(men) > 0:
7      m = men[-1]
8      w = P_m(m)[-1]
9      if w not in matched_W:
10         matched_M[m] = w
11         matched_W[w] = m
12         del P_m(m)[-1]
13         del men[-1]
14     else: # if w is already matched
15         m' = matched_W[w]
16         # and prefers m to m'
17         if P_w[w][m] > P_w[w][m']:
18             # match m with w
19             matched_M[m] = w
20             matched_W[w] = m
21             del P_m(m)[-1]
22             del men[-1]
23             # unmatch m'
24             del matched_M[m']
25             matched_M.append(m')
```

Part 2. Sorting and Order Statistics

11. Heaps

Array Heaps[1] are a data structure built on top of an array A, i.e. a structural invariant and a collection of functions that maintain that invariant. Heaps come in two flavors: Min heaps

[1] Heaps can be built on top of trees.

and Max heaps. The invariant for a Max heap is $A[i] \leq A[\lfloor i/2 \rfloor]$. Furthermore each entry has "children": $A[2i]$ is the left child and $A[2i+1]$ is the right child of element $A[i]$.

11.1. **Max Heapify.** To re-establish the heap property we use a procedure that fixes violations by switching the violator with its largest child and then recursing. Running time is $O(\lg n)$.

```
Max-Heapify(A, i)
1  largest = i
2  # if the left child exists and is greater then potentially switch
3  if 2i ≤ len(A) and A[2i] > A[i]:
4      largest = 2i
5  # if the right child exists and is greater then switch
6  if 2i + 1 ≤ len(A) and A[2i + 1] > A[largest]:
7      largest = 2i + 1
8  A[i], A[largest] = A[largest], A[i]
9  # potentially fix violation between child and one of its children
10 Max-Heapify(A, largest)
```

11.2. **Build Max Heap.** To build a heap from an array notice that the deepest children/leaves are already legal heaps so there's no need to `Max-Heapify` them, and the children start at $\lfloor \text{len}(A)/2 \rfloor$. Running time is $O(n)$.

```
Max-Heapify(A)
1  for i = ⌊len(A)/2⌋ downto 1:
2      Max-Heapify(A, i)
```

11.3. **Extract Min.** $A[1]$ is the maximum element in the heap (by the Max heap invariant), but removing it isn't as simple as just popping it off the top since the invariant might be violated. It's also not as simple as simple as replacing $A[1]$ with it's largest child because. The solution is to replace with the last element in the heap and then re-establish the invariant. Running time is $O(\lg n)$.

```
Extract-Min(A)
1  m = A[1]
2  A[1] = A[−1]
3  A.pop()
4  Max-Heapify(A, 1)
5  return m
```

11.4. **Heap sort.** You can use `Extract-Min` in the obvious way to sort an array. Running time is $O(n \lg n)$.

```
HeapSort(A)
1  s = [ ]
2  while len(A) > 0:
3      s.append(Extract-Min(A))
4  return reversed(s)
```

11.5. **Heap increase key.** In various instances you might want to increase the position of a key in the heap, such as when each key corresponds to the priority of some task. This just involves re-establish the Max heap invariant by "percolating" the entry up the array. Running time is $O(\lg n)$.

```
Heap-Increase-Key(A, i, key)
1  if key < A[i]:
2      throw Exception(key is smaller than current i key)
3  A[i] = key
4  # if child is bigger then parent then swap
5  while i > 1 and A[⌊i/2⌋] < A[i]:
6      A[⌊i/2⌋], A[i] = A[i], A[⌊i/2⌋]
7      i = ⌊i/2⌋
```

11.6. **Heap insert.** Using `Heap-Increase-Key` we can insert into the heap by insert and $-\infty$ element at the end of the heap and then increasing the key to what we want. Running time is $O(\lg n)$

```
Max-Heap-Insert(A, key)
1  A.append(−∞)
2  Heap-Increase-Key(A, len(A), key)
```

12. Quicksort

Quicksort is experimentally the most efficient sorting algorithm. The randomized version runs in $O(n \lg n)$ but is typically faster. It works by dividing and conquering.

```
Quicksort(A)
if len(A) ≤ 1:
    return A
else:
    # randomly pick a pivot
    p = Random(1, len(A))  # inclusive
    # swap so that you can exclude from contention the pivot
    A[p], A[−1] = A[−1], A[p]
    # partition
    A_left = filter(A[: −1], λe : e ≤ A[−1])
    A_right = filter(A[: −1], λe : e > A[−1])
    # recursively sort
    A_left = Quicksort(A_left)
    A_right = Quicksort(A_right)
    # combine
    return A_left + A[−1] + A_right
```

13. Counting Sort

The lower bound on sorting in the comparison model (i.e. using comparisons as an ordering relation) is $\Theta(n \lg n)$. But if one doesn't use comparisons then $\Theta(n)$ is possible. Counting sort is one such $\Theta(n)$ algorithm. If keys range from 1 to k in $A = [a_1, \ldots, a_n]$ then counting sort counts the number of keys less than or equal to each key a_i and then places a_i in that position.

```
Counting-Sort(A)
k = max(A)
C = (k + 1) * [0]
# count how many of values from 1 to k there is
for i = 1 to len(A):
    C[A[i]] = C[A[i]] + 1
# count how entries in A less or equal to i
for i = 1 to k:
    C[i] = C[A[i]] + 1
# now place the items in the correct places
B = (k + 1) * [None]
# go in reverse direction in order for sort to be stable
for i = len(A) downto 1:
    # a_i has C[a_i] elements to its left in B
    B[C[A[i]]] = A[i]
    # if there are multiples of a_i then the next
    # should be left of in order for stable
    C[A[i]] = C[A[i]] - 1
```

14. Radix Sort

Radix sort use the same technique that casinos use to sort cards (apparently?): sort stably least significant to most significant digit. For n numbers in base d where each digit ranges from 1 to k the running time is $\Theta(d(n+k))$ if the stable sort runs in $\Theta(n+k)$.

```
Radix-Sort(A, d)
for i = 1 to d:
    # let's pretend i can pass Insertion-Sort a key
    Insertion-Sort(A, key=lambda a : a[-i])
```

15. Bucket Sort

Bucket sort depends on values being uniformly distributed $[0, 1]$. It buckets all the entries and then subsorts. Expected run time is $\Theta(n)$.

```
Bucket-Sort(A)
n = len(A)
B = n · [[ ]]
for i = 1 to n:
    # bucket (imagine n = 10).
    # the +1 is because ⌊10 (0.01)⌋ = 0
    B [⌊nA [i]⌋ + 1] .append (A [i])
for i = 1 to n:
    Insertion-Sort(B [i])
return B
```

16. Order statistics

16.1. **Quickselect.** Any sorting algorithm can be used to compute kth order statistics: simply sort and return the kth element. But using the ideas of `Quicksort` you can get down to expected time $O(n)$: only recurse to one side.

```
Quickselect(A, k)
if len(A) ≤ 1:
    return A [1]
else:
    p = Random (1, len (A)) # inclusive
    A [p] , A [−1] = A [−1] , A [p]
    A_left = filter (A [: −1] , λe : e ≤ A [−1])
    if k == len (A_left + 1)
        # in sorted order A [1 : len (A_left + 1)] = A_left + [A [−1]]
        # and so the pivot is 1 "in front" of A_left
        return A [−1]
    elif k < len (A_left)
        # the kth order statistic in A is still the kth order statistic in A_left
        return Quickselect(A_left, k)
    else:
        A_right = filter (A [: −1] , λe : e > x)
        # the kth order statistic is (k − len (A_left) − 1) th statistic in A_right
        # think about it likes this: A = [1, 2, 3, 4, 5] and we partition on
        # 3 and we look for the 4th order statistic. well obviously it's
        # 4 = A_right [k − len (A_left) − 1] = A_right [1]
        return Quickselect(A_right, k − len (A_left) − 1)
```

16.2. **Quickerselect.** Using *median-of-medians* in order to guarantee good splits we can get down to $O(n)$ worst case (not just expected).

```
Quickerselect(A, k)
if len(A) == 0:
    return A[1]
else:
    # divide into n groups of 5 (except for the last one)
    # and use a sort in order to get medians.
    n = ⌊len(A)/5⌋
    m_1 = Insertion-Sort(A[1 : 5 + 1])[3]
    m_2 = Insertion-Sort(A[5 : 10 + 1])[3]
    ⋮
    m_n = Insertion-Sort(A[5n :])[⌊⌊len(A[5n:])/2⌋⌋]
    # recursively compute median of medians and use it as the pivot
    # after this recursive call the pivot is in position ⌊n/2⌋
    Quickerselect([m_1, m_2, ..., m_n], ⌊n/2⌋)
    A[⌊n/2⌋], A[−1] = A[−1], A[⌊n/2⌋]
    x = A[−1]
    A_left = filter(A[: −1], λe : e ≤ x) + [x]
    if k == len(A_left)
        return x
    elif k < len(A_left)
        return Quickselect(A_left, k)
    else:
        A_right = filter(A[: −1], λe : e > x)
        return Quickselect(A_right, k − len(A_left))
```

Part 3. **Data Structures**

17. Hash Tables

Hash tables are m length arrays keyed on strings instead of numbers.

17.1. **Hash function.** A Hash function is something that "hashes" up strings into numbers. It should uniformly distribute the keys over the hash space, meaning each key k is equally likely to hash to any of the m slots of the hash table. A good hash function according to Knuth is

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

where $A \approx \left(\sqrt{5}-1\right)/2$ and $kA \mod 1$ means the fractional part of kA, i.e. $kA - \lfloor kA \rfloor$.

17.2. **Hashing with chaining.** Hashing with chaining is basically Bucket Sort, except with the $\lfloor \rfloor$ replaced by a Hash function and retrieval.

Hashing with Chaining

```
1  HashInsert(H, k, v)
2  # H is hash table, k is key, v is value
3      m = len(H)
4      hsh = ⌊m (k (√5 − 1) /2  mod 1)⌋
5      H[hsh].append(v)
6
7  HashRetrieve(H, k)
8      m = len(H)
9      hsh = ⌊m (k (√5 − 1) /2  mod 1)⌋
10     if len(H[hsh]) > 0:
11         return H[hsh][1]
12     else:
13         return None
14
15 HashDelete(H, k, v)
16     m = len(H)
17     hsh = ⌊m (k (√5 − 1) /2  mod 1)⌋
18     i = 1
19     while i ≤ len(H[hsh]):
20         if H[hsh][i] == v:
21             del H[hsh][i]
22             return
23         else:
24             i = i + 1
25     return "Error: v not in table"
```

If n is the total number of items in the hash table and m is the length of the hash table then on average (give uniform hashing) each list has $\alpha = n/m$ items. Therefore insertion is $\Theta(1)$, and retrieval/deletion is $\Theta(1+\alpha)$.

17.3. **Hashing with open addressing.** In hashing with open addressing the buckets are "linearized", i.e. just laid out in the table itself: inserts and searches hash and then traverse forward in the table until they find a spot. Deletion is harder so if deletion is necessary then hashing with chaining should be used. Insertion costs at most $1/(1-\alpha)$ and for $\alpha < 1$ retrieval costs

$$\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$$

Integral to these bounds is that α the load factor stay small. In order for the amortized analysis to workout the hash table should be doubled in size (and entries copied) when the table becomes full but halve it only when the load goes down to below 1/4.

Hashing with open addressing

```
HashInsert(H, k, v)
# H is hash table, k is key, v is value
    m = len(H)
    hsh = ⌊m (k (√5 − 1) /2  mod 1)⌋
    hsh_orig = hsh
    if H[hsh] == NIL:
        H[hsh] = (k, v)
    else:
        hsh = hsh + 1
        while H[hsh] ≠ NIL and hsh ≠ hsh_orig:
            # mod so it swings back around and +1
            # because indexing starts at 1, not 0
            hsh = (hsh + 1  mod m) + 1
        if H[hsh] == NIL:
            H[hsh] = (k, v)
        else:
            retun "Error: Hash table full"

HashRetrieve(H, k)
    m = len(H)
    hsh = ⌊m (k (√5 − 1) /2  mod 1)⌋
    hsh_orig = hsh
    if H[hsh][1] == k:
        return H[hsh][2]
    else:
        hsh = hsh + 1
        while H[hsh][1] == k and H[hsh] ≠ NIL and hsh ≠ hsh_orig:
            # mod so it swings back around and +1
            # because indexing starts at 1, not 0
            hsh = (hsh + 1  mod m) + 1
        if H[hsh] == NIL or hsh == hsh_orig:
            return "Error: key missing"
        else:
            return H[hsh][2]
```

18. Binary Search Tree

A binary tree is a graph where each vertex has at most two children. A binary search tree is a tree with the further constraint that the key of a parent is greater or equal to any of the keys in its left subtree and less than or equal to any of the keys in its right subtree.

The working low-level data structure for trees is dict().

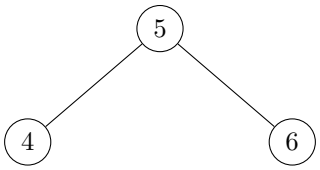
Binary Tree

```
btree = lambda parent, name, val, lchild, rchild:
            {'parent':parent, 'name':name, 'val':val,
             'lchild':lchild, 'rchild':rchild}

# counter generator is for labeling
def counter(x):
    start = x
    while True:
         yield start
         start += 1
c = counter(1)

root = btree(None,next(c),5,None,None)
left = btree(root,next(c),4,None,None)
right = btree(root,next(c),6,None,None)
root['lchild'] = left
root['rchild'] = right
```

Note that the `name is purely` a label and has no relation to `val`. The tree then looks like



18.1. **Inserting into a binary search tree.** Inserting into a binary search tree is easy: the insert vertex just has to obey the binary search constraint: start at the root, if the root value is equal to the key you're inserting then go left, otherwise go right. Once you hit a `None` create a new vertex. Running time is $O(\lg n)$ if the tree is balanced.

```
Binary-Tree-Insert(B, k)
# B is a btree dict as described above, corresponding to the
# root of the tree
prnt = ptr = B
while ptr ≠ None:
    # because of python's memory model we need to keep track
    # of parent since names are references not pointers, i.e.
    # you can't reassign pointers like in C
    prnt = ptr
    if ptr['val'] ≤ k:
        ptr = ptr['lchild']
    else:
        ptr = ptr['rchild']
if prnt['val'] ≤ k
    prnt['lchild'] = btree(prnt, next(c), k, None, None)
    return prnt['lchild']
else:
    prnt['rchild'] = btree(prnt, next(c), k, None, None)
    return prnt['rchild']
```

18.2. **Searching a binary search tree.** Searching is easy because a binary search tree obey the binary search constraint: start at the root, if the root value is equal to the key you're searching for then you're done, otherwise if the key is less than the value go left, otherwise go right. Running time is $O(\lg n)$ if the tree is balanced.

```
Binary-Tree-Search(B, k)
ptr = B
while ptr ≠ None and ptr['val'] ≠ k:
    if ptr['val'] < k:
        ptr = ptr['lchild']
    else:
        ptr = ptr['rchild']
if ptr == None:
    return "Error: key missing"
else:
    return ptr
```

18.3. **Binary search tree min/max.** The minimum of a binary tree is the left-est most vertex. Running time is $O(\lg n)$ if the tree is balanced.

```
Binary-Tree-Min(B)
1 ptr = B
2 while ptr['lchild'] ≠ None:
3     ptr = ptr['lchild']
4 return ptr
```

The maximum of a binary tree is the right-est most vertex. Running time is $O(\lg n)$ if the tree is balanced.

```
Binary-Tree-Max(B)
1 ptr = B
2 while ptr['rchild'] ≠ None:
3     ptr = ptr['rchild']
4 return ptr
```

18.4. **Binary search tree predecessor/successor.** The predecessor of a vertex the maximum of a vertex's left subtree. Running time is $O(\lg n)$ if the tree is balanced.

```
Binary-Tree-Predecessor(B, k)
1 ptr = Binary-Tree-Search(B, k)
2 return Binary-Tree-Max(ptr['lchild'])
```

The successor of a vertex the minimum of a vertex's right subtree. Running time is $O(\lg n)$ if the tree is balanced.

```
Binary-Tree-Predecessor(B, k)
1 ptr = Binary-Tree-Search(B, k)
2 return Binary-Tree-Min(ptr['rchild'])
```

18.5. **Deleting from a binary search tree.** We need an auxiliary function to wrap up some code that's re-used. Running time is constant.

```
Transplant(u, v)
1  # does not handle case where u is root of tree
2  prnt = u['parent']
3  if prnt['lchild'] == u
4      prnt['lchild'] = v
5  else:
6      prnt['rchild'] = v
7  if v ≠None:
8      v['parent'] = prnt
```

Deleting from a binary search tree is a little more complicated. Since the binary search tree property needs to be always preserved it's unclear what to replace a deleted vertex with. A child? A parent? In fact it should be the successor (or predecessor). The successor is the vertex whose value would follow the vertex you're trying to delete if you listed all the vertices in order. How do you find the successor? It's the minimum of the right subtree or the vertex (and the minimum of a tree is the farthest left of the tree). Then that minimum can be replaced by it's right child (it has no left child). Running time is $O(\lg n)$ if the tree is balanced.

```
Binary-Tree-Delete(B, k)
ptr = Binary-Tree-Search(B, k)
# trivial case, successor is parent
if ptr['rchild'] == None:
    Transplant(ptr, ptr['lchild'])
elif ptr['lchild'] == None:
    Transplant(ptr, ptr['rchild'])
else:
    succ = Binary-Tree-Successor(ptr)
    # if the successor is the right child of ptr then
    # then right child has no left child and task simple
    if succ == ptr['rchild']:
        Transplant(ptr, succ)
        succ['lchild'] = ptr['lchild']
        succ['lchild']['parent'] = succ
    else: # otherwise we have to fix successor subtrees and do the same thing
          # including fixing the right child
        # fix successor
        Transplant(succ, succ['rchild'])
        # don't lose right child of ptr
        succ['rchild'] = ptr['rchild']
        succ['rchild']['parent'] = succ
        # move successor into ptr's position
        Transplant(ptr, succ)
        succ['lchild'] = ptr['lchild']
        succ['lchild']['parent'] = succ
```

18.6. **Pre-order/In-order/Post-order traversal.** A Pre-order/In-order/Post-order traversal of a binary tree is a traversal the manipulates the vertex either before left and right children, after the left child but before the right child, and after both the left and right children. The easiest way to implement any of these is recursion but iterative versions do exist. Running time is $O(n)$ since the traversal visits every vertex. For illustrative purposes we simply print the `val` attribute, but any operation on the vertex could be performed.

```
Pre-order-traversal(B)
print(B['val'])
Pre-order-traversal(B['lchild'])
Pre-order-traversal(B['rchild'])
```

```
In-order-traversal(B)
1  Pre-order-traversal(B['lchild'])
2  print(B['val'])
3  Pre-order-traversal(B['rchild'])
```

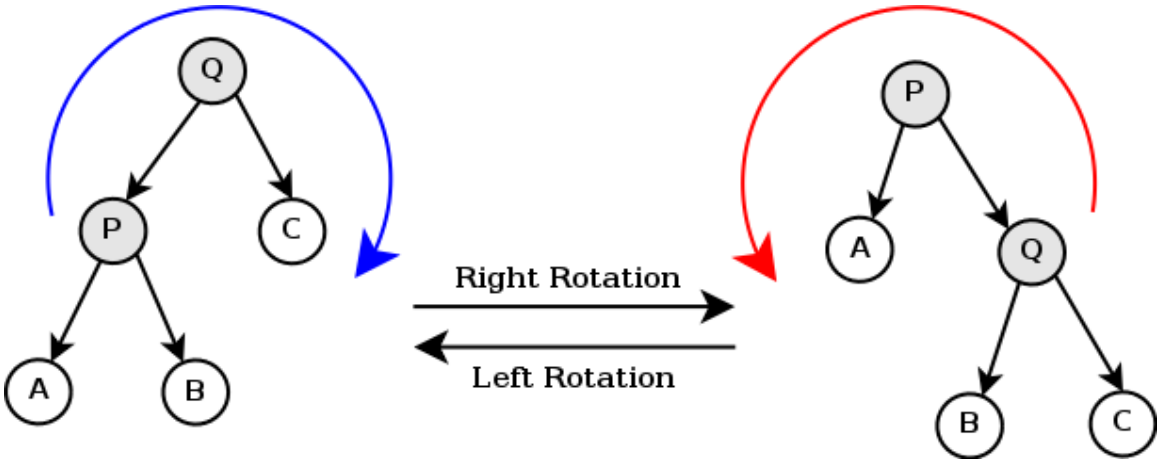
```
Post-order-traversal(B)
1  Pre-order-traversal(B['lchild'])
2  Pre-order-traversal(B['rchild'])
3  print(B['val'])
```

19. Treap

Binary trees have $O(\lg n)$ queries and inserts and deletions if they're balanced. Turns out keep them balanced is tough - a ton of schemes exist. The simpliest is a random binary tree using a treap. A treap combines the invariants of a binary tree *and* and a heap. There are two sets of attributes: priorities and keys. The priorities obey the heap property (children have smaller priority than their parents) and the keys obey the binary search property. In order to get a balanced binary tree, which is the value of treaps, we randomly generate a priority key. This then simulates the generation of a random binary tree which on average has depth $O(\lg n)$. We use a min heap.

19.1. **Treap search.** Just like for binary search tree and hence omitted.

19.2. **Treap insert.** This is easier of the two operations. First we need two auxiliary functions `Left-Rotate` and `Right-Rotate`. The easiest way to remember these is pictures



```
Left-rotate(p)
p_prnt = p['parent']
a = p['lchild']
q = p['rchild']
# put q in p's position
if p = p_prnt['lchild']:
    p_prnt['lchild'] = q
else:
    p_prnt['rchild'] = q
p['rchild'] = q['lchild']
q['lchild'] = p
```

```
Right-rotate(p)
q_prnt = q['parent']
p = q['lchild']
c = q['rchild']
# put p in q's position
if q = q_prnt['lchild']:
    q_prnt['lchild'] = p
else:
    q_prnt['rchild'] = p
q['lchild'] = p['rchild']
p['rchild'] = q
```

To insert into a treap, generate a random priority, and insert the key as if it were a binary search tree (i.e. at the bottom), then rotate up until the heap property is restored.

```
Treap-Insert(T, k)
# T is a binary tree that's a treap
u = Random(0, 1) # both ends inclusive
ptr = Binary-Tree-Insert(T, (u, k))
prnt = ptr['parent']
while prnt ≠ None and ptr['val'] < prnt['val']:
    if ptr == prnt['lchild]:
        Right-Rotate(prnt)
    else:
        Left-Rotate(prnt)
    ptr = prnt
    prnt = ptr['parent']
```

19.3. **Treap delete.** To delete a vertex rotate it down until it's a leaf node and then delete the leaf node. Rotate down according to which of the vertex's children have a higher priority: if the left child has a higher priority than the right then rotate right, otherwise rotate left.

```
Treap-Delete(T, k)
# T is a binary tree that's a treap

ptr = Binary-Tree-Search(T, k)
while ptr['lchild'] ≠ None or ptr['lchild'] ≠ None:
    if ptr['lchild'] ≠ None and ptr['lchild']['val'] > ptr['rchild']['val']
        Right-Rotate(ptr)
    else:
        Left-Rotate(ptr)
    if ptr = ptr['parent']['lchild']:
        ptr['parent']['lchild'] = None
    else:
        ptr['parent']['rchild'] = None
    del ptr
```

20. Cartesian Tree

Given a sequence of **distinct** numbers (or any totally ordered objects), there exists a binary min-heap whose inorder traversal is that sequence. This is known as the Cartesian tree for that sequence. A min-treap is an easy way to construct a Cartiesian tree of a sorted sequence. Why? Obviously: it's is heap ordered since it obeys the min heap property and an in order traversal reproduces the sequence in sorted order. How to construct a Cartesian tree for an arbitrary sequence $A = [a_1, \ldots, a_n]$? Process the sequence values in left-to-right order, maintaining the Cartesian tree of the nodes processed so far, in a structure that allows both upwards and downwards traversal of the tree. To process each new value x, start at the node representing the value prior to x in the sequence and follow the path from this node to the root of the tree until finding a value y smaller than x. This node y is the parent of x, and the previous right child of y becomes the new left child of x. Running time is $O(n)$.

```
Cartesian-Tree(A)
T = btree(None, next(c), A[1], None, None)
ptr = prnt = T
for i = 2 to len(A):
    while prnt['parent'] ≠ None and A[i] < prnt['val']
        prnt = prnt['parent']
    if prnt == None: # then we're at the root
        # and A[i] is the smallest value we've seen so far
        ptr = btree(None, next(c), A[i], prnt, None)
        prnt['parent'] = ptr
    else:
        ptr = btree(prnt, next(c), A[i], prnt['rchild'], None)
        prnt['rchild'] = ptr
    prnt = ptr
```

21. Interval Trees

An interval tree is built atop your favotire balanced binary tree data structure (treap in our case) and stores left endpoints as key. It also keeps track of maximum right endpoint in the subtree rooted at a vertex. It supports interval intersection tests (very useful). Maintaining the max in insertion and deletion is straightforward.

21.1. **Interval search.** Interval search works by being optimistic: $i = [a, b]$ and $j = [x, y]$ two intervals overlap if either $a \leq x \leq b$ or $x \leq a \leq y$. Therefore at each interval we test for overlap and whether $x \leq a \leq y$ where y is the maximum right endpoint for any interval in the left subtree. If so we go left. If in fact $y < a$ then no interval in the left subtree could possibly intersect so we go right.

```
Interval-Tree-Search(T, i)
# T is an interval tree, i = [a, b]
a, b = i[1], i[2]
# j['left'] is left endpoint of interval and
# j['right'] is right endpoint
intersect = lamba j: a ≤ j['left'] ≤ b or j['left'] ≤ a ≤ j['right']
# 'int' is interval associated with vertex
while T ≠ None and not intersect(T['int']):
    if T['lchild'] ≠ None and a ≤ T['lchild']['max']:
        T = T['lchild']
    else:
        T = T['rchild']
return T
```

22. Union-Find

A union-find data structure is a data structure suited for taking unions and finding members (duh). The particular units of the data structures are sets (not hash table derivatives), each with a representative. The data structure is very thin, basically a wrapper for the primitive data, except for a pointer to the representative of the set and two heuristics that speed up the operations. The path compression heuristic "compresses" the path to representative of the set by setting it to be equal to that representative (which it might not be after a union). The weighted union heuristic makes it so that the smaller of the two sets unioned is the one whose representative pointers need to be updated.

Amortized complexity of n `Make-Set`, `Find-Set, Union,` operations where m are `Make-Set` is $O(m\alpha(n))$ where $\alpha(n)$ is the Ackermann function and $\alpha(n) \leq 4$ for any realistic application.

```
Make-Set(x)
1  return {'val':x,'rep':x,'rank':0}
```

22.1. Make set.

22.2. Find set.

Find set is interesting: it unwinds the stack in order to reset all the representatives from x to the representative of the set.

```
Find-Set(x)
1  if x['rep'] ≠ x:
2      x['rep'] = Find-Set(x['rep'])
```

22.3. Union.

Find set is interesting: it unwinds the stack in order to reset all the representatives from x to the representative of the set. Running time is $O(m \lg n)$.

```
Union(x, y)
1  x_rep = Find-Set(x)
2  y_rep = Find-Set(y)
3  if x_rep['rank'] > y_rep['rank']
4      y_rep['rep'] = x_rep
5  else:
6      x_rep['rep'] = y_rep
7      if x_rep['rank'] == y_rep['rank']
8          # it's an approximate rank.
9          y_rep['rank'] = y_rep['rank'] + 1
```

23. Euler circuit

An Euler circuit visits each vertex in graph twice - once going past it and once coming back across it. How do you print out an Euler circuit of a tree? Use a modified depth first traversal.

```
Euler-Circuit(u)
# u is a vertex with children
# print it going past
print(u)
for v in u['children']:
    Euler-Circuit(v)
    # print is coming back
    print(u)
```

24. Tarjan's Least Common Ancestor

The least common ancestor w of two vertices u, v in a tree is the ancestor common to both that's of greatest depth. The algorithm is useful for range-minimum querying. It uses the same traversal as `Euler-Circuit` and the Union-Find data structure augmented with a property `ancestor`. The algorithm proceeds by growing "bottom up" sets corresponding to subtrees whose roots are the least common ancestors of any pair of vertices in the tree **which have been completely traversed by the Euler circuit**. Let P be a global with the set of vertices you're interested in finding least common ancestor of and initialize all vertices to have `color` Blue in order to represent unfinishined (i.e. not completely traversed by the Euler circuit).

```
Least-Common-Ancestor(u)
# u is the root of a tree
u_set = Make-Set(u)
#this is the Euler-Circuit transformation (equivalent of print)
u_set['ancestor'] = u
for v in u['children']:
    Least-Common-Ancestor(v)
    # let's pretend there's a big table where i can fetch v_set from
    Union(u_set, v_set)
    u_set['ancestor'] = u
# u_set['val'] = u
u_set['val']['color'] = Red
for each v such that {u, v} ∈ P:
    if v['color'] == Red:
        print("Least common ancestor of {u, v} is " + v_set['ancestor'])
```

25. Range Minimum Queries

Given a sequence of distinct values and a subsequence (specified by it's end points) what is the minimum value of the in that subsequences? It's just the least common ancestor of the end points in the cartesian tree representing the sequence.