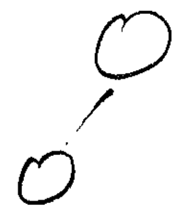
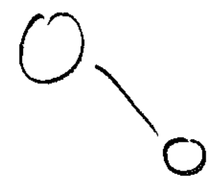
Height of AVL Trees:

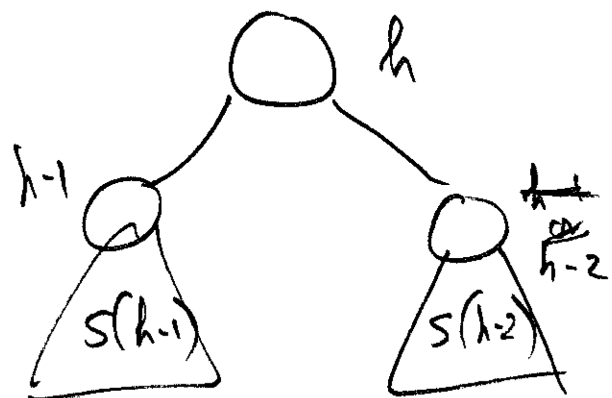
Let $S(h)$ = Min # of nodes in an AVL tree of height h.

$S(0) = 1$ 

$S(1) = 2$  or 

~~St~~ $S(h) =$

$S(h-1) + S(h-2) + 1$



$$(S(h)+1) = (S(h-1)+1) + (S(h-2)+1)$$

$$f(h) = f(h-1) + f(h-2)$$

$f(0) = 2 \qquad f(1) = 3$

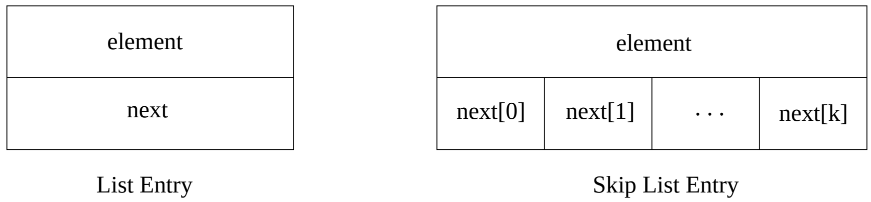
Fibonacci! starts are different.

$$f(h) = \Theta(c^h) \qquad c = \frac{\sqrt{5}+1}{2} \approx 1.618$$

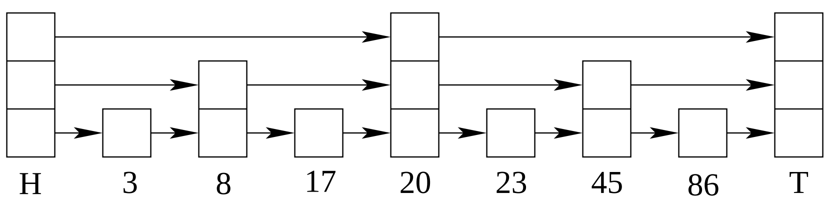
A tree with n nodes has height $O(\log_c n) = O(\log n)$

Skip Lists

Generalization of sorted linked lists for implementing Dictionary ADT (insert, delete, find, min, succ) in $O(\log n)$ expected time per operation. Skip lists compete with balanced search trees like AVL, Red-Black, and B-Trees.



The elements are stored in sorted order, in a linked list of nodes. Each skip list entry has an array of next pointers, where $next[i]$ points to an element that is roughly 2^i nodes away from it. The *next* array at each entry has random size between 1 and $maxLevel$, the maximum number of levels in the current skip list. Ideally, $maxLevel \approx \log n$. Each skip list has dummy head and tail nodes, both of $maxLevel$ height, storing sentinels $-\infty$ and $+\infty$, respectively. Iterating through the list using $next[0]$ will go through the nodes in sorted order. A reference to the previous element can also be stored by adding a *prev* field to Skip List Entry.



Search starts at the top level, goes as far as possible at each level, without going past target, descending one level at a time, until reaching the target node. Addition/Removal of nodes makes it difficult to maintain an ideal skip list, in which $next[i]$ of a node points to a node that is exactly 2^i away from it. Skip lists solve this problem by selecting the number of levels (size of $next[\ ]$) of a new node probabilistically.

Skip List implementation:

```
Entry class:
    T element
    Entry[ ] next
    Entry prev  // prev is optional
    int[ ] span  // for indexing

Entry(x, lev):  // constructor
    element ← x
    next ← new Entry[lev]
    span ← new int[lev]

SkipList class:
    Entry head, tail  // dummy nodes
    int size, maxLevel
    Entry[ ] last  // used by find()
    Random random

SkipList(): // Constructor
    head ← new Entry(null, 33)  // sentinel −∞, with maximum number of levels
    tail ← new Entry(null, 33)  // sentinel +∞, with maximum number of levels
    size ← 0
    maxLevel ← 1
    last ← new Entry[33]
    random ← new Random()

find(x):  // helper method to search for x.  Sets last[i] = node at which search came down from level i to i-1
    p ← head
    for i ← maxLevel-1 down to 0 do
        while p.next[i].element < x do    // watch out for NPE because of null element in tail
            p ← p.next[i]
        last[i] ← p

contains(x):  // is x there in list?
    find(x)
    return last[0].next[0].element == x

remove(x):  // delete x
    if not contains(x) then return null
    ent ← last[0].next[0]
    for i ← 0 to ent.next.length-1 do
        last[i].next[i] ← ent.next[i]   // bypass ent at level i
    size ← size - 1
    return ent.element
```

```
add(x):  // insert x
        if contains(x) then return false  // reject duplicate
        lev ← chooseLevel()   // length of next[ ] for x's entry
        ent ← new Entry(x, lev)
        for i ← 0 to lev-1 do
                ent.next[i] ← last[i].next[i]
                last[i].next[i] ← ent
        ent.next[0].prev ← ent;  ent.prev ← last[0]   // if prev link is defined in Entry
        size ← size + 1
        return true

chooseLevel():  // Prob(choosing level i) = 1/2 Prob(choosing level i-1)
        // Slow method:
        lev ← 1
        while random.nextBoolean() do lev++   // should limit to 33
        if lev > maxLevel then maxLevel ← lev
        return lev

        // fast method:
        lev ← 1 + Integer.numberOfTrailingZeros(random.nextInt())
        // Optionally, lev ←  min(lev, maxLevel+1), to allow maxLevel to grow gradually
        if lev > maxLevel then maxLevel ← lev
        return lev
```

Indexing in skip lists:

```
get(index):  // return element at index (first element is at index 0)
        if index < 0 or index > size - 1 then throw NoSuchElementException
        p ← head
        for i ← 0 to index do
                p ← p.next[0]
        return p.element
```

Running time of get(index) is O(1+index), which is O(n) in the worst case.

Improving the RT of get(): Each Entry stores the span of next[i] in span[i] (number of elements between the two nodes). Code for find() has to be updated to calculate the indexes of the nodes in last[]. Code for add(), and remove() need to be modified to update the appropriate span[] values to reflect the changes to the list made by the operations.