Search algorithms

1. ## Basic search algorithms
   - Linear search: $A_{LSearch}(N) \sim \frac{2N}{\log{(N)}}$ $W_{LSearch}(N) = N$ $B_{LSearch}(N) = 1$
   - Binary search: $A_{BSearch}(N) \sim \log(N) + O(1)$ $W_{BSearch}(N) = \lceil \log{(N)} \rceil$ $B_{BSearch}(N) = 1$

In a search <u>decision tree</u> with N internal nodes, the **lower bound** is:

$W_A(N) \geq height_{min}(N)$ or $W_A(N) \geq \Omega(\log{(N)})$

$A_A(N) \geq \Omega(\log{(N)})$

For all algorithms A that are based in key comparisons.

We can conclude that given this bounds **binary search is optimal**.

2. ## Search on dictionaries
A dictionary <u>is a sorted set of data</u> that supports the following operations:
   - <u>Search</u>(key, dict), returns the position of the key in the dictionary.
   - <u>Insert</u>(key, dict), inserts the key in the dictionary .
   - <u>Remove</u>(key, dict), removes the key in the dictionary.

We have several options for the data structure of the dictionary:

   a. **Sorted array**, the <u>search is optimal</u> but the <u>insertion is costly</u> (Θ(N) very bad).
   b. **Binary search Tree**. In this structure the parent node is bigger than his left child but smaller than his right child.
   The <u>cost of a search</u> in a binary search tree is <u>O(height(T))</u>.
   The <u>cost of the insertion</u> in a binary search tree is <u>O(height(T))</u> (being T the tree).
   The <u>cost of removing</u> in a binary tree is <u>the cost of searching a key and readjusting</u> the tree.

There are 3 cases when we remove a key:

   1. The node we want to remove <u>doesn't have any child</u>, we just simply remove the child.
   2. The node we want to remove <u>has one child</u>, we take that child and we place it in the place were the removed node was.
   3. The node <u>has two childs</u>, we need to find a successor, for this we move once to the right and then we keep moving to left until we find a node without any left child.

So, searching <u>in a binary search tree</u> has the following cost (searching in a binary search tree is not the same as doing a binary search, a binary search can be performed in an array):

$$A_{Search}^{S}(N) = 1 + \frac{1}{N} A_{Create}(N)$$

$$A_{Create}(N) = 2N \cdot log(N) + O(N)$$

$$A_{Search}^{S}(N) = \Theta(\log{(N)})$$

3. AVL Trees

An AVL tree is a binary search tree in which in every subtree, the balance factor is BF(T')={-1,0,1}. A balance factor in a node is defined as <u>the height in his left subtree – the height in his right subtree</u>.

In order to build this tree, we follow 2 steps, first we perform the normal insertion in a binary search tree, and second, if it is needed, we rebalance the node with a rotation.

Unbalance	Rotation
(-2,-1)	Left Rotation (LR) at -1 (Left child of -1 turns into right child of -2)
(2,1)	Right rotation (RR) at 1 (Right child of 1 turns into left child of 2)
(-2,1)	Right-left rotation (RLR) at the left of 1 RR(left(1))+ LR(left(1))
(2,-1)	Left-right rotation (LRR) at the right of -1 LR(right(-1))+ RR(right(-1))

If T is an AVL with N nodes, then height(T)=Θ(log(N)). All primitives related with a binary search tree which is an AVL have a log(N) runtime.

Removing in an AVL tree is different, the common solution is to perform a **lazy deletion**, the node to remove is marked as free.
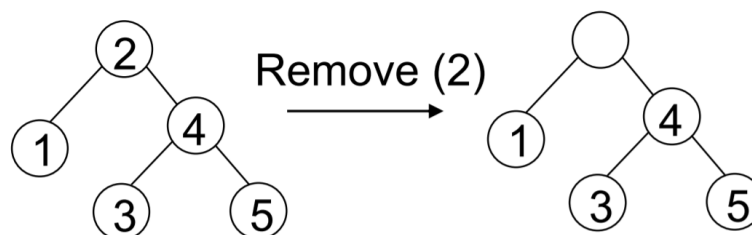
Mimimum AVLs

h	AVL	n_h	n_h+1	F_{h+2}
0	•	1	2	F_2
1		2	3	F_3
2		4	5	F_4
3		7	8	F_5
4		12	13	F_6
…			….	….

A Fibonacci tree is an AVL tree with the minimum number of nodes for a specific height n. We can see that if we add the number of nodes + number of nodes + 1 for a specific height in a tree T_n, we can see that the result of that addition is the number of nodes +1 of the tree with one node more of height T_{n+1}

4. Hashing.

With hashing we can search with a cost which is less than O(log(N)).

We have a dictionary with **data D**, each data has a unique **key k=k(D)**, so we search **by keys** but **not through keys**.

We have several ways of implementing hashing:

a. We calculate k*=max{k(D): D $\in$ D}, we store each D in an array T of size k* (assuming that there are not repeated keys).
   With this the cost is O(1), but if k* is too large then the amount of memory to store the array is excessive.

b. We fix M > |D| and we define an injective function where if $k \neq k'$ then $\mathbf{k}(k) \neq \mathbf{k}(k')$ so $\mathbf{k}:\{k(D)/D \in D\} \rightarrow \{1,2,3,\ldots,M\}$.
   Now we place D at the index $\mathbf{k}(k(D))$ in the array T.
   So, searching is done in a constant time with a reasonable memory consumption. The problem is that it is very hard to find such **injective and universal function**.
c. We search for a universal **k** universal function (valid for any set of keys), and we allow that **k** is not injective. Now it is possible to have **collisions** because $k \neq k'$ and $\mathbf{k}(k) = \mathbf{k}(k')$, so we have to implement a mechanism to deal with collisions and create a hash function.

When we create hash functions, our goal is to have the less possible collisions. So, we have an array T with M data, this means that **p(collision)=1/M**. One way for implementing this function is to use random indexes using rand().

Another way is using divisions, so $m > |D|$ where **m** is prime, then we define **h(k)=k%m**.

Another way is using multiplications, so $m > |D|$ and we have Φ as an irrational number equal to $(1 + \sqrt{5})/2$ or $(\sqrt{5} - 1)/2$, then we define $\mathbf{h(k)} = \lfloor m \cdot (k \cdot \Phi) \rfloor$

Let's now define a **uniform hash function**. A hash function is uniform if given k and k' where k≠k', then p(h(k) = h(k')) = 1/m

In order to solve collisions, we do it in these ways:

- **Chaining**: If we have a collision, we create a linked list.
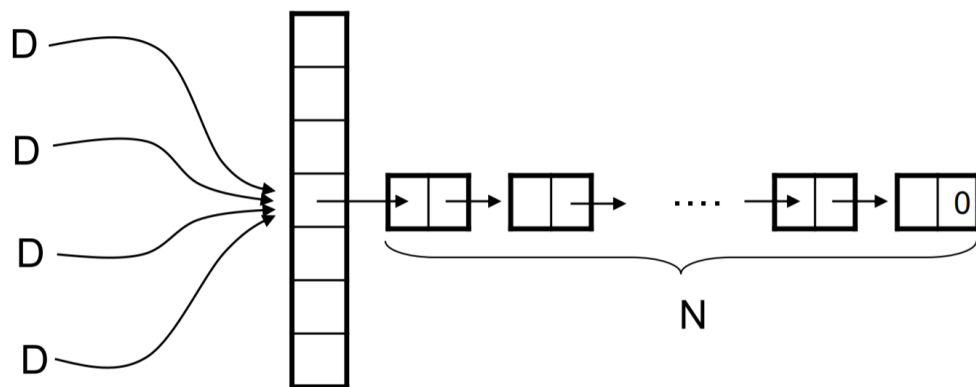
   The search on average is:
     In the **failure** case: $A_{SHC}^{f}(N,m) = \frac{N}{m} = \lambda$
     In the **successful** case: $A_{SHC}^{s}(N,m) = 1 + \frac{\lambda}{2} + O(1)$

- **Open addressing**: We have several methods in open addressing, but they are all related with **probing**, which is that if the place we want to store the key is occupied we search for another place.
     In **linear probing**, if we have a collision, we store the data in the next free space. p=T[h(D)], (p+1)%m if this is occupied then (p+2)%m,… until we find one free.
$$A_{LP}^{f}(N,m) = \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right) \quad and \quad A_{LP}^{s}(N,m) = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$$

     In **quadratic probing**, we do the same as in linear probing but in positions $(p+1^2)\%m,\ldots$
     In **random probing**, we try random positions. In case we use random probing, we have that:
$$A_{HRP}^{f}(N,m) = \frac{1}{1-\lambda} \quad and \quad A_{HRP}^{f}(N,m) = \frac{1}{\lambda}log\frac{1}{1-\lambda}$$

The Average cost in probing methods is:

     In the **failure case** $A_{P}^{f}(N,m) = f(\lambda)$

In the **successful case** $A_P^S(N, m) \cong \frac{1}{\lambda} \int_0^\lambda f(u) du$