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: $A_{LSearch}(N) \sim \log(N) + O(1)$ $W_{LSearch}(N) = \lceil \log(N) \rceil$ $B_{LSearch}(N) = 1$

In a search decision tree 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

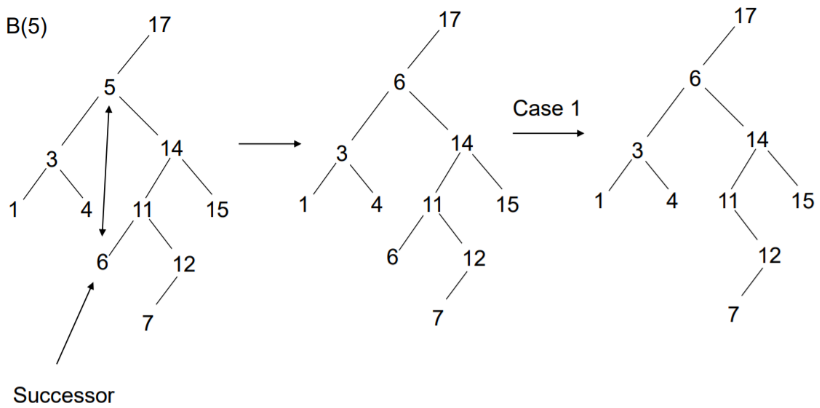
A dictionary is a sorted set of data that supports the following operations:

- Search(key, dict), returns the position of the key in the dictionary.
- Insert(key, dict), inserts the key in the dictionary .
- Remove(key, dict), removes the key in the dictionary.

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

a. **Sorted array**, the search is optimal but the insertion is costly (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 cost of the insertion in a binary search tree is O(height(T)) (being T the tree).
   The cost of removing in a binary tree is the cost of searching a key and readjusting the tree.

This is an example of removing and readjusting:



So, searching in a binary search tree has the following cost:

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

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

$$A^S_{Search}(N) = \Theta(\log(N))$$

3. AVL Trees

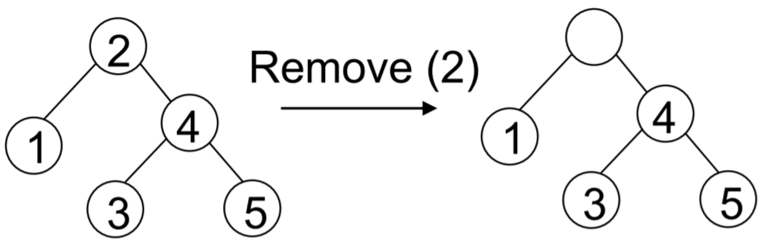
An AVL tree is a data structure based in a binary search tree which uses **balance factors** in each node. A balance factor in a node is defined as the height in his left subtree – the height in his right subtree, this balance can only be: -1, 0 and 1.

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)=O(log(N)).

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



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 ∈ $\mathcal{D}$}, 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 > $|\mathcal{D}|$ and we define an injective function where if $k \neq k'$ $then$ $\boldsymbol{k}(k) \neq \boldsymbol{k}(k')$ so **k**:{k(D)/D∈ $\mathcal{D}$}→{1,2,3,…,M}.
   Now we place D at the index **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 $\boldsymbol{k}(k) = \boldsymbol{k}(k')$, 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().

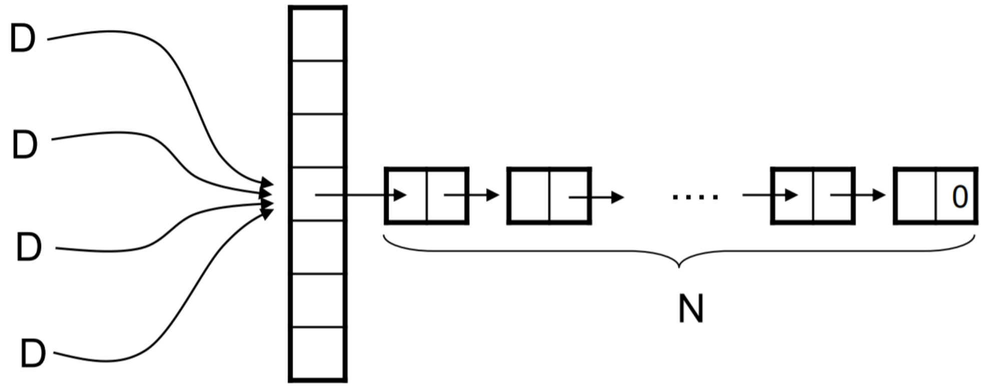
Another way is using divisions, so $\boldsymbol{m} > |D|$ where **m** is prime, then we define **h(k)=k%m**.

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

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: $A^{f}_{SHC}(N,m) = \frac{N}{m} = \lambda$

  In the **successful** case: $A^{s}_{SHC}(N,m) = 1 + \frac{\lambda}{2} + O(1)$

- **Open addressing**: We have several methods in open addressing.

  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.

  In **quadratic probing**, we do the same as in linear probing but in positions $(p+1^2)$%m,…

  In **random probing**, we try random positions.

The Average cost in probing methods is:

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

In the **successful case** $A^{s}_{P}(N,m) \cong \frac{1}{\lambda}\int_{0}^{\lambda} f(u)du$