DATA STRUCTURES

GEORGE SALMAN

Contents

Introduction.

Abstract. In the following part we are going to discuss different data structures which are already familiar to us E.g array, Linked lists, and complexity of each one, what are pros and cons of using the following data structures, and how they used for solving problems, Moreover, we are going to disscuss how array are allocated in memory. A way of representation polinomyals by cyclic linked list and unique way to represent sparseness matrices.

Array, Matrices, Linlked lists.

Array as a abstract data structure, the operation are:

(1) The operation $create(type, i)$ return a array of elements with type and indexes inn the set I.
(2) The operation $store(A, i, e)$ add elements e to the array elemnts i.e $A[i] = e$.
(3) The operation $get(A, i)$ return the elements which were inserted in index i lately.

Remark. All the values are with the same type, and they are all determined when we create it with operation *create*.

> What happen when we are try to insert element with different type?

Solution. can't determine, in some cases we got error,and on other we get default value 0, and in other cases we get a garbage value.

> what happen when we call from index which nothing were inserted to it?

Solution. it depends on out implementation, some time we require to intialize the array with 0 to all of it elements, we don't require that for our to definition.

Implementation s to array:

Proposition. *(1). a continious place in memory.*

- the execution time of *get* and store is $O(1)$.
- the time required for creating new array depend on the system, and to the condition of we are asked to fill all the array with intialize value while creating.

Proposition. *(2). a number of continious places in memory each one in size n.*

in order to implement array with size m, first we need to choose $k = [m/n]$ and the element $A[i]$ exists in the address $base[[\lfloor i/n \rfloor] + imodn$.

Example. if we choose $n = 6$ and $i = 16$ first in order to find memory of $A[16]$ it's exist in the place $base[\lfloor 16/6 \rfloor + 16\%6 = base[2] + 4$, and the computing time of the address is $O(1)$.

Remark. every element in the base array is a pointer to the first address of the on of thus subarrays Moreover, it's important that subarrays satisfy the same length for simple compution advantage.

n dimentional array:

Proposition. *continious place in memory*

Example. we can implemeny a 2 dim array as a continious memory, assuming that *base* is the momery of $A[0][0]$ then, the memory of $A[i]$ is $base + i \cdot n$ and the $A[i][j]$ is $base + i \cdot n + j$.

Example. a three dim array we can find the memory of $A[i_1][i_2][i_3]$ by the following formula:

$$base + i_3 \cdot n_2 \cdot n_1 + i_2 \cdot n_1 + i_1$$

We can generalize the following to n dim i.e the memoy of $A[i_d]....[i_1]$ in $A[n_d]...[n_1]$ is computed by the formula:

$$base + i_d \cdot n_{d-1} \cdots n_1 + i_{d-1} \cdot n_{d-2} \cdots n_1 + ... + i_1$$

$$= base + i_d \prod_{i=1}^{d-1} n_i + i_{d-1} \cdot \prod_{i=1}^{d-2} n_i + ... + i_1$$

$$\text{base}+((..(i_d n_{d-1} + i_{d-1})n_{d-2} + i_{d-2}) + n_{d-3} + ... + i_3)n_2 + i_2)n_1 + i_i$$

we can look at the folwoing as polinom,

$$p(x_0) = a_k x^k + a_{k-1}x^{k-1} + .. + a_1 x + a_0 =_* (...(a_k x + a_{k-1})x + ... + a_1)x + a_0$$

the form $*$ help us reduce number of mulitiplication from d^2 to d.

```
for (i=0, i<n, i++)
    for (j=0, j<n, j++)
        A[i][j] = 0;
```

```
for (i=0, i<n, i++)
    for (j=0, j<n, j++)
        A[j][i] = 0;
```

which code of the following scan array more quick?

Solution. notice that the right code check line by line so it take more time than the right side Since, we know that in oprian A we scan the first place then it's easier to move to the memory because we have continious place in memory so we got easier access i.e by the operating system proberties.

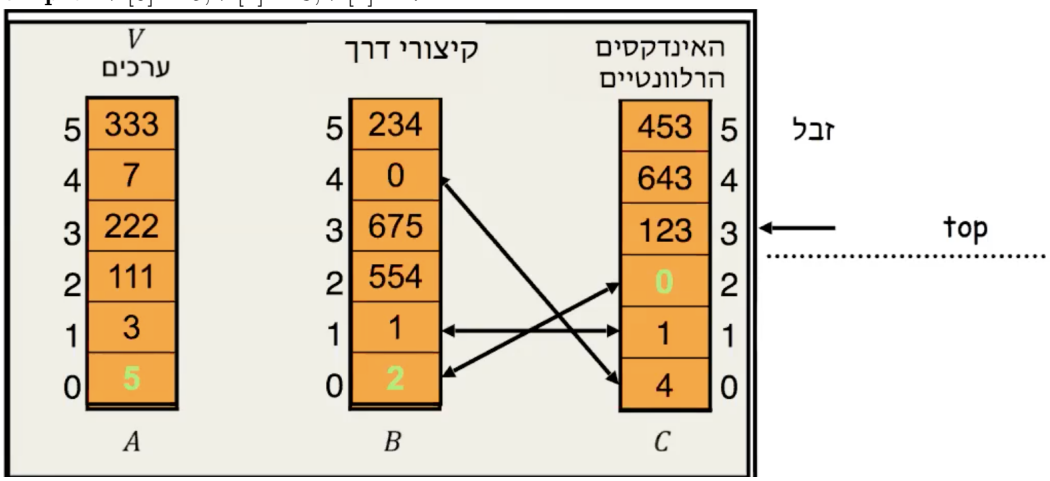
Remark. in the following course, both implementation are equivalent in meaning of complexity, we are not getting into high resolution of how operating system works.

is it possible to intialize array in $O(1)$?

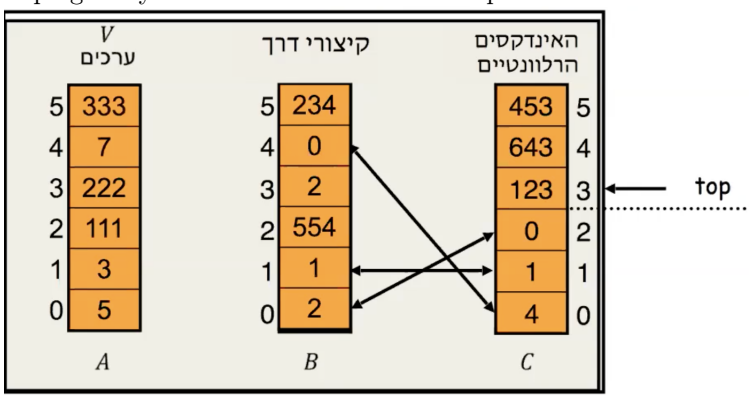
Solution. YES.

first we can allocate memory to new array, in order to implement the follwing, first we use two arrays, the first array is our regular array and the other array give us the indexes we can access.

Example. $V[0] = 5, V[1] = 3, V[4] = 7$.



notice that we put our array include the indexes in which we want to access E.g $0, 1, 4$ and in order to remember them we put pointer top from the place in which all the wanted indexes are finished, now we create two additional arrays assuimig B, C s.t array C remember our indexes in which we need to intialize and array B is helping array which has the in the values place of C number of index i.e



now we can define a function which check if index is intialized:

```
int  is_initialized(int i) {
        return ( B[i] < top  &&  B[i] >= 0  &&  C[B[i]] = = i );
}
```

E.g we need to check if element 5 is intialized, we see that $B[5] \not< top$ hence. it's not intialized, identical way we can check that the 3 element is not intialized since $C[B[3]] = 2 \neq i = 3$ when for example the 4 element satisfy that $B[4] = 0 \leq top = 3 \wedge B[4] \geq 0 \wedge C[B[4]] = 4 = i = 4$

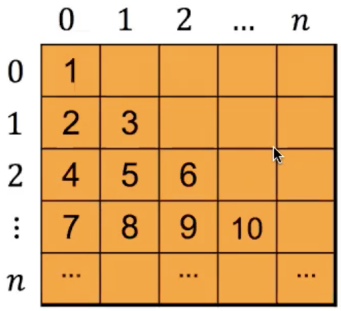
Reperesentation address of symmetric matrices:

Proposition. *collection of vectors which are stored in continious array E.g* 1 23 456 78910

Given a symmetric matrix A which imply $A[i, j] = A[j, i], \forall i, j$.

we can represent the address of $A[i,j]$ by the following formula:

$$1+2+3...+i=\frac{i(i+1)}{2}$$



the follwoing formula represent number of the element appeared in previous line E.g in the follwoing photo we can see that before line 2 appeared $1+2=3$ elements, hence, it's easier to access a memory by the follwoing formula by adding a j then we can access the wanted element in following line.

i.e we can define:

$$Addr(i,j)=\left\{\begin{array}{cc} Base+\frac{i(i+1)}{2}+j, & i\geq j \\ Add(j,i) & otherwise \end{array}\right\}$$

sparseness matrix:

Definition. a sparseness matrix is a two dimentianl array in which almost of the elements have constant value or a paramater c.

Definition. (notation meaning).

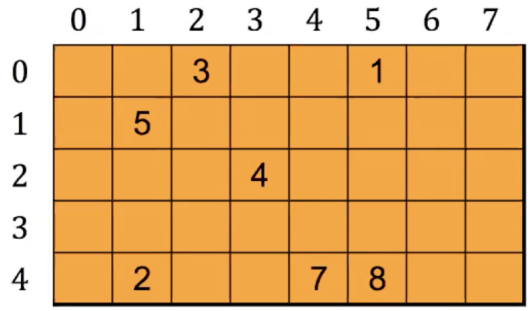
a sequence of matrix $M_1,..,M_n$ which satisfy that the number of elements which are not constant is with order $o(m(n))$ when $m(n)$ is the maximal number of places in matrix M_n thus are a sequence of **sparseness matrix.**

Example. diagonal matrices.

Remark. in **sparseness matrix** the number of elements which are not constant c is $o(n^2)$.

Representation:

a list of threes (i,j,A_{ij}) ordered lexicographically.



Example. the following matrix:
represented as following:
$(0,2,3),(0,5,1),(1,1,5),(2,3,4),(4,1,2),(4,4,7),(4,5,8)$

pros and cons:
advantange:

(1) save memory for a **sparseness matrices.**
(2) velocity of adding and multiplication operation for **sparseness matrices.**

disadvanteges:

(1) there is no random acceess in $O(1)$ time.

Adding matrices:

adding two matrices with order nxn take $O(n^2)$ time in general case, now we assume that we have two matrices s.t in the first matrix we have m elements are not constant c and in the other matrix k. the addition of thus matrices is by merging two list of the follwing matrices. the time of merging is there total length which is $O(m+k)$ i.e for **sparseness matrices** the time for adding them is $o(n^2)$.

Linked lists:
pros and cons:
advantange:

(1) dynamic memory allocating.
(2) scanning elements in linear time

disadvanteges:

(1) there is no random acceess in $O(1)$ time.
(2) there is need to do small allocate for each time we insert element.

Init
intializing operation : init(head)

```
void  init (NODE  *head){
          * head = NULL;
}
```

searching
searching operation: find(x,head)

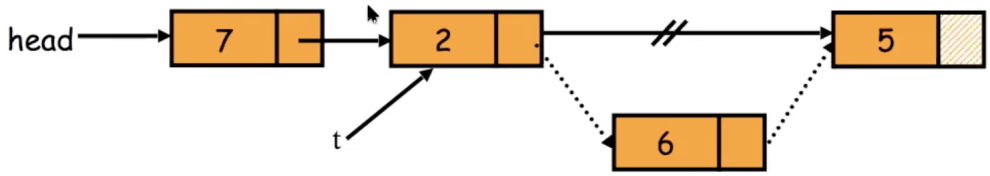
```
NODE * find (DATA_TYPE x, NODE  *head){
      NODE *t;
      t = head;
      while (t!=NULL && t → info != x)
            t = t → next ;
      return t;
}
```

in the worst case it take $O(n)$.

Insert element

insert element operation: insert(t,x)

```
int  insert ( NODE  *t,  DATA_TYPE x){
      NODE *p;
      if (t == NULL) return 0;
      p = (NODE *) malloc (sizeof (NODE));
      p → info = x;
      p → next = t → next ;
      t → next  = p ;
      return 1;
}
```



Technion

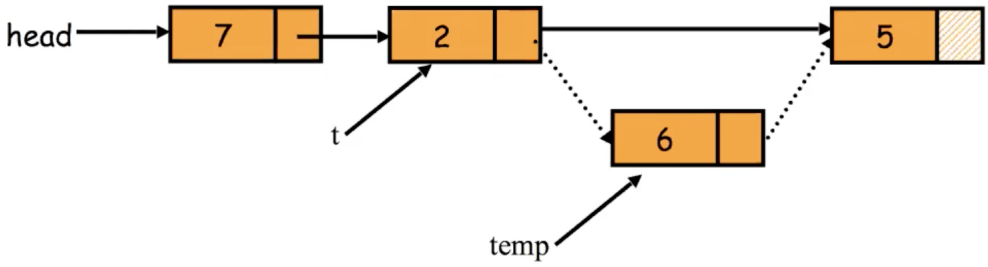
it take $O(1)$ to insert element.

Delete

delete operation: delete(t).

the pointer t point to the vertice before the vertice in which we want to delete.

```
delete ( NODE  *t){
          NODE  * temp ;
          temp = t → next;  /*מצביע לצומת שמורידים */
          t → next = temp → next ;
          free (temp) ;
}
```

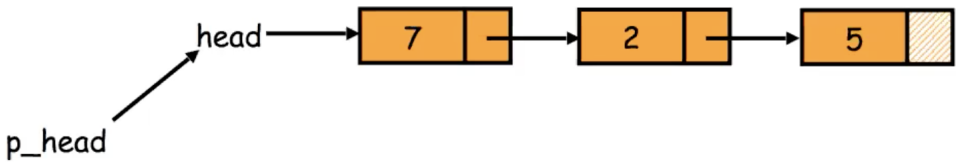


it take $O(1)$ to delete element.

Deleting first element

deleting first operation: delete_first(r).

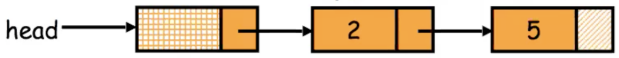
```
int delete_first ( NODE  **p_head){
        NODE  * temp ;
        if (*p_head == NULL)  return 0 ;
        temp = *p_head;
        *p_head = temp → next ;
        free (temp) ;
        return 1;          /* success code
    */
}
```



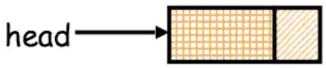
in this implementation three is private case for the first element, so we want to avoid that by adding empty element in the beginning of the linked list which is called "dummy node" this addition save us:

(1) a private code for empty lists.
(2) private code to delete the first node.
(3) private code to add first node.

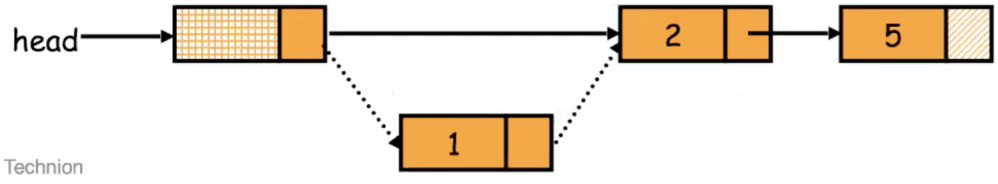
in this represnentation our linked list will look as following;



and empty list would look as following:



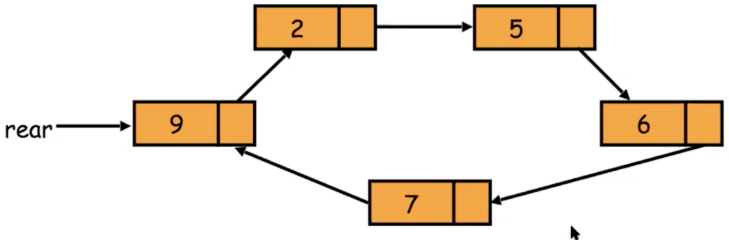
and inserting element before the first element is identical to inserting regular element:

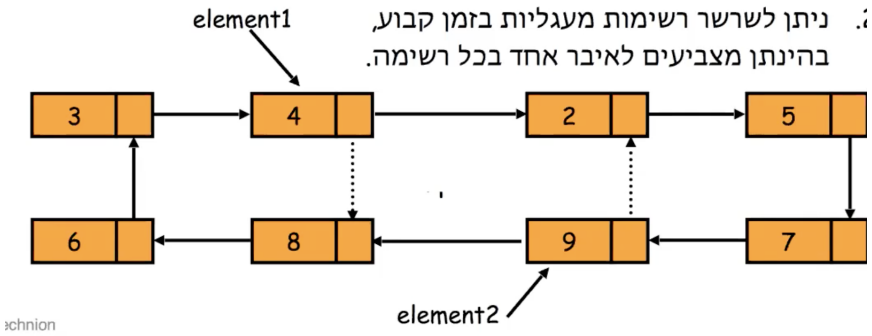


Cyclic Linked Lists:

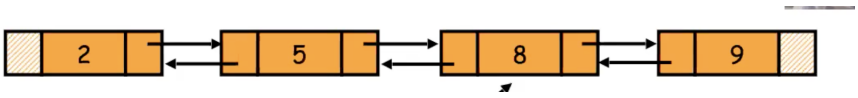
advantages:

(1) we can acceess from element to any element.
(2) we can cahin cyclic linked lists in contstant time, given on pointer on every one of the linked list.



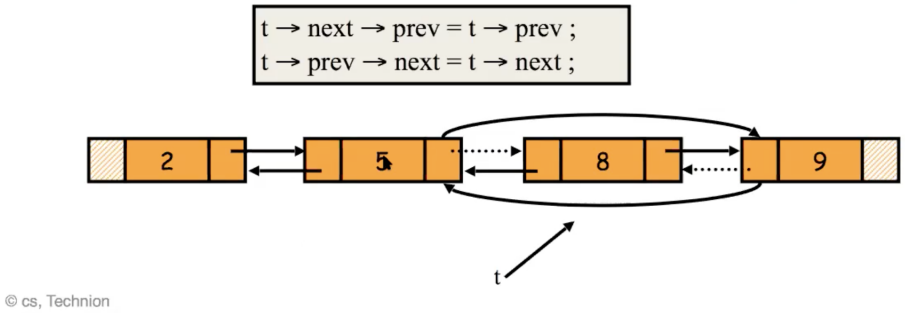


Two directianal Linked Lists:

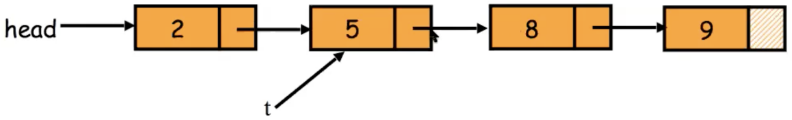


Advanatages:

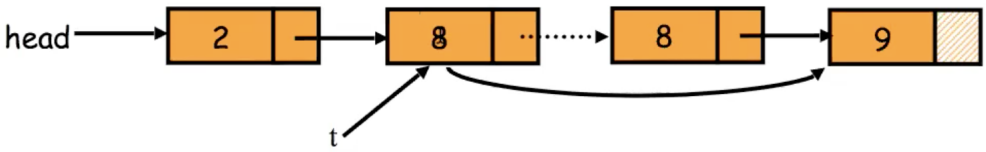
(1) we can from element to any elements.
(2) we can delete element given a pointer t on it.



can we delete element which we have pointer t on it aslo in one directional linked list?



we can simply, duplicate the information in element before t to the element in which t point too, then we delete the element before t.

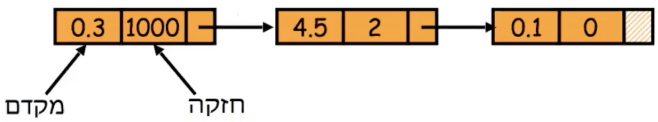


Disadavanatages:

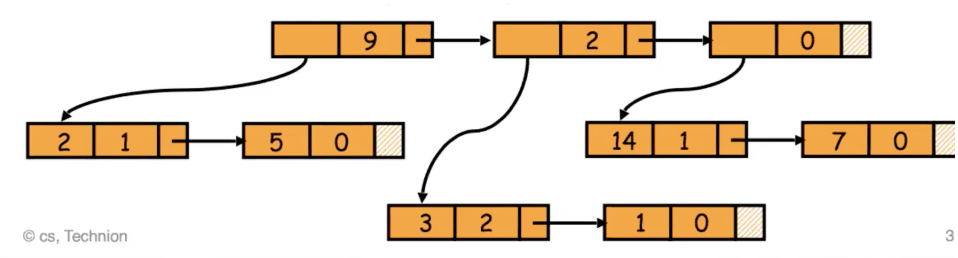
(1) doesn't work for the last element.
(2) may there be addition pointer to the element which were pointed too (8) and there is no way to update thus pointers.
(3) may there additional pointer to the element we changed (5) and they expect to see 5 there/
(4) a node could includ a lot of information, and copying would take time.

Representation of Sparseness Polynomials:

E.g one variable would be preseneted as following:



we can represenet a polinomyals which there coefficients are also polinomylas by a cyclic linked lists:



Notations & Complexity.

Why we need complexity boundries?

- How to compare between different algorithms?
- In case we didn't have a complexity boundries, how can we compare between algorithms?
- We could run on computer and check how much time take each algortihm, and how much it require.
- But in which input we want to use? for sorting algorithms to array with length n, we have $n!$ different inputs., so algorthim 1 can work better for specefic input while the other won't.

Definition. (big O).
We say that $f(n) = O(g(n))$ if exist constants $c, n_0 > 0$ s.t $\forall n \geq n_0$:

$$f(n) \leq c \cdot g(n)$$

Example. Simple examples:

- $f(n) = g(n) = n$.
- $f(n) = n, g(n) = n^2$.

Definition. (big Omega Ω).
We say that $f(n) = \Omega(g(n))$ if exist constants $c, n_0 > 0$ s.t $\forall n \geq n_0$:

$$f(n) \geq c \cdot g(n)$$

Definition. (big Theta Θ).
We say that $f(n) = \Theta(g(n))$ if $f(n) = \Omega(g(n)) \wedge f(n) = O(g(n))$.

Exercise. let $P(n)$ polinomyal with positive coefficients and degree k i.e

$$P(n) = a_0 + a_1 n + ... + a_k n^k = \sum_{i=0}^{k} a_i n^i, a_k > 0$$

(1) Show that $P(n) = O(n^k)$.
(2) Show that $P(n) = \Omega(n^k)$.

Solution. (1).
We need to show that exist constants $c, n_0 > 0$ s.t $\forall n \geq n_0$ s.t $P(n) \leq n^k$.
Notice that:

$$P(n) = \sum_{i=0}^{k} a_i n^i \leq \sum_{i=0}^{k} a_i \cdot n^k = (\sum_{i=0}^{k} a_i) \cdot n^k$$

So we define $c = \sum_{i=0}^{k} a_i$ and we are finished.

Solution. (2).
We need to show that exist constants $c, n_0 > 0$ s.t $\forall n \geq n_0$ s.t $P(n) \geq n^k$.
Notice that:

$$P(n) = \sum_{i=0}^{k} a_i n^i \geq a_k \cdot n^k$$

So we define $c = a_k$ and we are finished.

Corollary. $P(n) = \Theta(n^k)$.

Exercise. Show that $\forall k \geq 2$ satisfied:

$$\sum_{i=0}^{k} k^i = O(k^n)$$

We need to show that exist constants $c, n_0 > 0$ s.t $\forall n \geq n_0$ s.t $\sum_{i=0}^{k} k^i \geq k^n$.
Notice that:

$$\sum_{i=0}^{k} k^i = \frac{k^{n+1} - 1}{k - 1}$$

Since $k \geq 2$ satisfied that $k - 1 \geq \frac{k}{2}$:

$$\sum_{i=0}^{k} k^i = \frac{k^{n+1} - 1}{k - 1} \leq \frac{k^{n+1}}{k - 1} \leq \frac{k^{n+1}}{\frac{k}{2}} = 2 \cdot k^n$$

So we can define $c = 2$.

Lemma. $log(n!) = \Theta(nlogn)$.

Remark. We we write logarithm without base, we mean base 2.

Remark. If we change base, does ir change the correctness of the term?
No, Since changing base is equivalet to multiplying by scalar i.e:

$$log_b n = log_a n \cdot \frac{1}{log_a b}$$

Proof. We will prove by the definitions:
First direction we can see that:

$$logn! = log(1 \cdot 2 \cdot \cdots \cdot n) = log1 + log2 + ... + logn \leq nlogn$$

Hence, denote $c = 1, n_0 = 1$ and we have that $logn! = O(nlogn)$.
Other direction:

$$logn! = log(1 \cdot 2 \cdot \cdots \cdot n) = log1 + log2 + ... + log\frac{n}{2} + ... + logn$$

$$\geq log\frac{n}{2} + ... + logn \geq \frac{n}{2} log\frac{n}{2}$$

$$\frac{n}{2} log\frac{n}{2} = \frac{n}{2}(logn - log2) = \frac{n}{2}(logn - 1) = \frac{n}{2}logn - \frac{n}{2} = *$$

$$* = \frac{n}{4}logn + \frac{n}{4}logn - \frac{n}{4} \cdot 2 = \frac{n}{4}logn - \frac{n}{4}(logn - 2)$$

$\forall n \geq 4$ satisfied that:

$$\frac{n}{4}logn - \frac{n}{4}(logn - 2) \geq \frac{n}{4}logn$$

Therefore, we can choose $c = \frac{1}{4}, n_0 = 4$.
In total we have that

$$logn! = \Omega(nlogn) \wedge logn! = O(nlogn)$$

□

Theorem about boundires compelxity. Given 4 functions $f_1(n), f_2(n), g_1(n), g_2(n)$ s.t:

$$f_1(n) = O(g_1(n))$$

$$f_2(n) = O(g_2(n))$$

Lemma 1. $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.

Proof. By the definition:
Exist constants $c_1, n_{01} > 0$ s.t $\forall n \geq n_{01}$:

$$f_1(n) \leq c_1 \cdot g_1(n)$$

Exist constants $c_2, n_{02} > 0$ s.t $\forall n \geq n_{02}$:

$$f_2(n) \leq c_2 \cdot g_2(n)$$

$$f_1(n) + f_2(n) \leq c_1 \cdot g_1(n) + c_2 \cdot g_2(n) \leq (c_1 + c_2) \cdot g_1(n) + (c_1 + c_2) \cdot g_2(n)$$

$$= (c_1 + c_2) \cdot (g_1(n) + g_2(n))$$

So we can let $c = c_1 + c_2, n_0 = max\{n_{01}, n_{02}\}$. □

-

Lemma 2. $f_1(n) + f_2(n) = O(max\{g_1(n), g_2(n)\})$.

Proof. Exist constants $c_1, n_{01} > 0$ s.t $\forall n \geq n_{01}$:

$$f_1(n) \leq c_1 \cdot g_1(n)$$

Exist constants $c_2, n_{02} > 0$ s.t $\forall n \geq n_{02}$:

$$f_2(n) \leq c_2 \cdot g_2(n)$$

$$f_1(n) + f_2(n) \leq (c_1 + c_2) \cdot (g_1(n) + g_2(n)) \leq (c_1 + c_2) \cdot 2max\{g_1(n), g_2(n)\}$$

So we can let $c = 2(c_1 + c_2), n_0 = max\{n_{01}, n_{02}\}$. □

Is it satisfied that $f_1(n)^{f_2(n)} = O(g_1(n)^{g_2(n)})$?

Solution. No, for counter example we can look at the following functions:

$$f_1(n) = n, f_2(n) = 2, g_1(n), g_2(n) = 1$$

and we got that $n^2 = O(n)$ which is not true. Since assume toward contradiction that it is true, i.e exist $c_1, n_{01} > 0$ s.t $\forall n \geq n_{01}$, $n^2 \leq c_1 \cdot n$ so we have that $n \leq c_1$ which is contradiction to the archemedian principle (i.e $\mathbb{N}$ not bounded).

running time of code.

Exercise. find complexity of the following code.

```
scanf("%d",&n);
k=2;
while(k<2){
k=k*2;
}
```

Solution. we can look at the table of iteration:

i	k
0	2
1	$2^2 = 4$
2	$2^3 = 8$
$\cdots$	$\cdots$
i	2^{i+2}
$\cdots$	$\cdots$
$log(n) - 1$	n

We can see that:

$$k = 2^{i+1} = n \Rightarrow i + 1 = log(n) \Rightarrow i = log(n) - 1$$

Exercise. find complexity of the following code.

```
scanf("%d",&n);
k=2;
while(k<2){
k=k*k*k;
}
```

Solution. we can look at the table of iteration:

i	k
0	2
1	$2^3 = 8$
2	$(2^3)^3 = 2^9 = 2^{(3)^2}$
$\cdots$	$\cdots$
i	$2^{(3)^i}$
$\cdots$	$\cdots$
$log(log(n))$	n

Since we can see that:

$$k = 2^{3^i} = n \Rightarrow 3^i = log_2(n) \Rightarrow i = log_3(log_2(n))$$

Definition. (small o).
We say that $f(n) = o(g(n))$ if $\forall c > 0$ exists $n_0 > 0$ s.t $\forall n \geq n_0$:

$$f(n) \leq c \cdot g(n)$$

Definition. (small o - equivalent definition).
We say that $f(n) = o(g(n))$ if:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Example. Look at:

- $f(n) = n, g(n) = n^2$.

Definition. (small omega ω).
We say that $f(n) = \omega(g(n))$ if $\forall c > 0$ exists $n_0 > 0$ s.t $\forall n \geq n_0$:

$$f(n) \geq c \cdot g(n)$$

Definition. (small omega ω - equivalent definition).
We say that $f(n) = \omega(g(n))$ if:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

Lemma. $\forall \epsilon > 0$ *satisfied* $log(n) = o(n^\epsilon)$.

Proof. We will show that following by the second definition of o small i.e

$$\lim_{n\to\infty} \frac{logn}{n^\epsilon} = \lim_{n\to\infty} \frac{\frac{1}{ln2} \cdot \frac{1}{n}}{\epsilon \cdot n^{\epsilon-1}} = \lim_{n\to\infty} \frac{1}{\epsilon \cdot ln2} \cdot \frac{1}{n^\epsilon} = 0$$

When the following stem from L'Hôpital's rule, and the fact the n^ϵ approach to infinit\y when $\epsilon > 0$. □

Exercise. given $f, g : \mathbb{N} \to \mathbb{R}$. prove/disprive:

Definition. $f(n) = S(g(n))$ if $f(n) = O(\sum_{i=1}^{n} g(i))$.

(1) $n^2 = S(n)$.

Solution. The following is true. we need to show that $n^2 = S(n)$ i.e $\exists c, n_0$ S.T $\forall n \geq n_0$:

$$n^2 \leq c \cdot \sum_{i=1}^{n} i = c \cdot \frac{n(n+1)}{2}$$

If we choose $c = 2$ then we get that $n^2 \leq n^2 + n$ which is true $\forall n \in \mathbb{N}$.

(a) $n = S(logn)$

Solution. The following is true. we need to show that $n = S(logn)$. i.e .e $\exists c, n_0$ S.T $\forall n \geq n_1$ satisfied:

$$n \leq c \cdot \sum_{i=1}^{n} logi = c \cdot log(n!)$$

we already saw that $log(n!) = \Theta(nlogn)$ so $\exists c', n_2$ S.T $\forall n \geq n_2$ satisfied:

$$c'nlogn \leq logn! \Rightarrow nlogn \leq \frac{1}{c'}logn!$$

So notice $\forall n \geq 2$ we have that $n \leq nlogn$ hence,

$$n \leq nlogn \leq \frac{1}{c'}logn!$$

so we can choose $n_0 = 2, c = \frac{1}{c'}$.

(i) if $f(n) = S(g(n))$ and $g(n) = S(h(n))$ then $f(n) = S(h(n))$.

Solution. The following is not true, for counter example take

$$f(n) = n^2, g(n) = n, h(n) = logn$$

then we have that $f(n) = n^2 = S(h(n)) = S(logn))$ which can't be true.

Exercise. if $n = O(f(n))$ then $\forall k \in \mathbb{N}$ $n = O(f^k(n))$, when $f^k(n)$ is operating f, k times on n.

Solution. The following is true, we are given that $n = O(f(n))$ therefore, $\exists c_1, n_1$ S.T $\forall n \geq n_1$ satisfied that $n \leq c_1 \cdot f(n)$. we will show by induction on k.

Base: for $k = 1$ it satisfied by the given $n = O(f(n)) = O(f^1(n))$.

Assumption: assuming that the following is true for k : $n = O(f^k(n))$, i.e $\exists c_k, n_k$ S.T $\forall n \geq n_k$ satisfied that $n \leq c_k \cdot f^k(n)$.

Step: we will show the following for $k + 1$.

Remark. notice that we want to get something as the following:

$$\underbrace{n \leq c_k \cdot f^k(n)}_{assumption} \leq \underbrace{c' \cdot f^{k+1}(n)}_{wanted}$$

so we try to show that:

$$f^k(n) \leq c' \cdot f^{k+1}(n)$$

by trying to use our induction base i.e the case $k = 1$ more specific,

$$f^k(n) \leq c_1 \cdot f(f^k(n))$$

in order to show that we need to show that:

$$n_1 \leq f^k(n)$$

First we will start to say something about $f^k(n)$, we will start by writing the given: induction assumption. it require us to assume something about n_{k+1} which is $n_{k+1} \geq n_k$. we need to show also that $n_1 \leq f^k(n)$,so we add to left inequality the same coeffiecient so we can vanish it. in order to do that we need to assume also that $n_{k+1} \geq c_k \cdot n_1$ so if we choose $n_{k+1} = max\{c_k \cdot n_1, n_k\}$ we have that:

$$c_k \cdot n_1 \leq n \leq c_k \cdot f^k(n) \Rightarrow n_1 \leq f^k(n)$$

$$\Rightarrow_* f^k(n) \leq c_1 \cdot f(f^k(n))$$

In total we choose $n_{k+1} = max\{c_k \cdot n_1, n_k\}, c_{k+1} = c_k \cdot c_1$ and get:

$$n \leq c_k \cdot f^k(n) \underset{*}{\leq} c_k \cdot c_1 \cdot f(f^k(n)) = c_{k+1} \cdot f^{k+1}(n) \Rightarrow n = O(f^{k+1}(n))$$

Remark. notice that we knew $n \leq c_1 \cdot f^k(n)$ and since we show that the induction is true we operate f^k on bothe side nd get that $f^k(n) \leq c_1 \cdot f^k(f(n))$, and by the induction assumption we got that $n \leq c_k \cdot f^k(n)$ so all together we got the result $n \leq c_k \cdot f^k(n) \underset{*}{\leq} c_k \cdot c_1 \cdot f(f^k(n)) = c_{k+1} \cdot f^{k+1}(n) \Rightarrow n = O(f^{k+1}(n))$.

Exercise. Prove/Disprove.

If $f(n) = \omega(1)$ then $\forall k \in \mathbb{N}$ we have that $f(n^{k+1}) - f(n^k) = \Theta((f^{k+1}(n))$.

Solution. By the definition, we have that $f(n)$ is not constant, so our intituition is to think about funciton which give us that $f(n^{k+1}) - f(n^k)$ is constant for some n but $f(n^{k+1})$ won't be, so we can take for counter example $f(n) = loglog(n)$.

First we need to check that $f(n)$ indeed is $\omega(1)$ i.e:

$$\lim_{n\to\infty} \frac{1}{loglog(n)} = 0 \Rightarrow loglog(n) = \omega(1)$$

We will show that the following is not true for $k = 1$:

$$f(n^{k+1}) - f(n^k) = f(n^2) - f(n) = loglog(n^2) - loglog(n)$$

$$= log(2 \cdot log(n)) - loglog(n) = log(2) + loglog(2) - loglog(2) = log(2)$$

Now notice that:

$$\lim_{n\to\infty} \frac{log(2)}{loglog(n^2)} = 0 \Rightarrow log(2) = o(loglog(n^2) \Rightarrow log(2) \neq \Theta(loglog(n^2))$$

other way (BY ME) is assuming toward contradiction that $log(2) = \Theta(loglog(n^2))$ hence, $\exists c, n_1$ S.T $\forall n \geq n_1$ we have that

$$loglog(n^2) \leq c \cdot log(2) = c \cdot log2 \Rightarrow log(2 \cdot log(n)) \leq c \cdot log2$$

$$log(2) + loglog(n) \leq c \cdot log(2) \Rightarrow loglog(n) \leq log(2)(c-1)$$

$$\Rightarrow n \leq (log(n)(c-1))^{10^{10}}$$

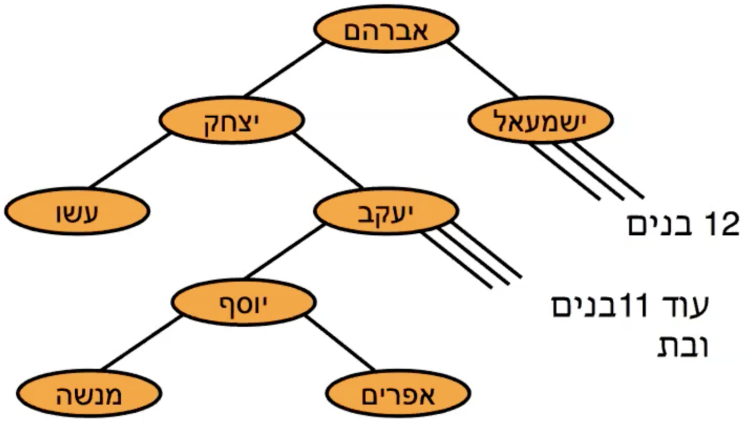
and it's a contradiction for the archemedian principle i.e $\mathbb{N}$ is not bounded.

Trees

Abstract. In the following chapter we are going to see data structures which we were not interduce to it before E.g AVL tree, binary trees, there implemenation, complexity of inserting elements to it, inserting algorithms, and special property of each one of thus trees. Moreover, how there implementation is applied in computer science to solve problems with low time complexity compared to other data structures E.g array, linked lists, etc.

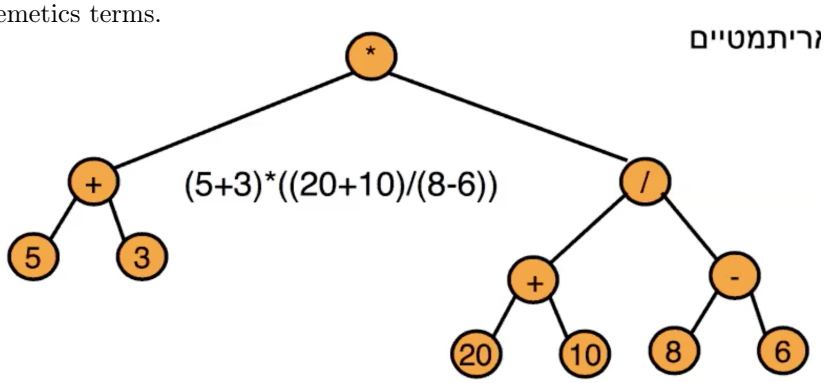
We can use trees for different uses E.g

- Genealogy.



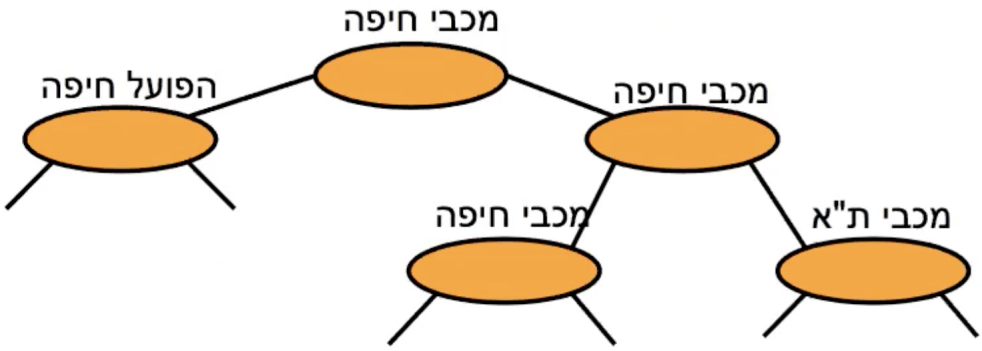
–

- Arithemetics terms.



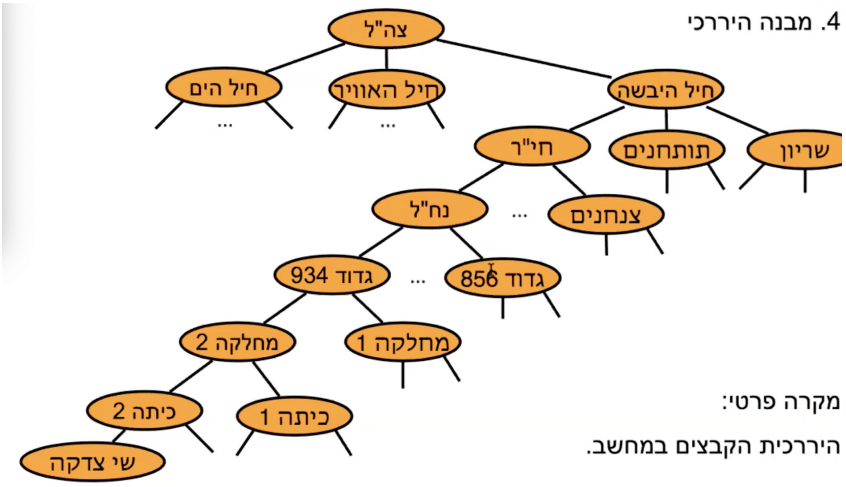
–

- Winners tree (soccer).



–

- Hierarchy Structure (E.g documents in PC).

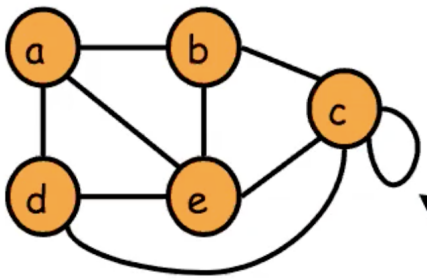


–

Undirected Graph.

Definition. Undirectied graph is a pair (V, E) which is contained by two sets vertices V and Edeges E. edge in E is a set of two elements from V, edge is maked by (i, j) (insead of accurated mark $\{i, j\}$).

1.1



In the following picture our sets are:

$$V = \{a, b, c, d, e\}$$

$$E = \{(a, a), (a, b), (a, d), (b, c), (b, e), (c, c), (d, c), (d, e), (e, c)\}$$

We denote $n = |V|, m = |E|$ in our example $n = 5, m = 9$.

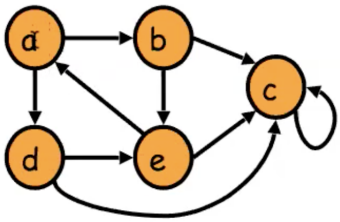
Remark. Number of edges m less in every graph than n^2 Since,

$$\binom{n}{2} + n = \frac{n(n-1)}{2} + n = \frac{n(n+1)}{2} < n^2$$

Directed graph.

Definition. Directed graph is a pair (V, E) which is contained by two set V and $E \subseteq VxV$.

1.2



$$V = \{a, b, c, d, e\}$$

$$E = \{(a,a), (a,b), (a,d), (b,c), (b,e), (c,c), (d,c), (d,e), (e,c)\}$$

Notice that the vertice which the arrow point too is represented as second coordination in the pair (E.g (e, c) notice that the represent the edge which point to vertice c and start from e).

Definition. directed path in directed graph (V, E) is a sequence of vertices $(v_1, ..., v_k)$ S.T for every pair of vertice which are sequential in the sequence, v_i, v_{i+1}, satisfied that (v_i, v_{i+1}) is edge in E. A directed path is called cycle if $v_1 = v_k$ (e.g in our example (a, d, e, a) is cycle path in the graph).

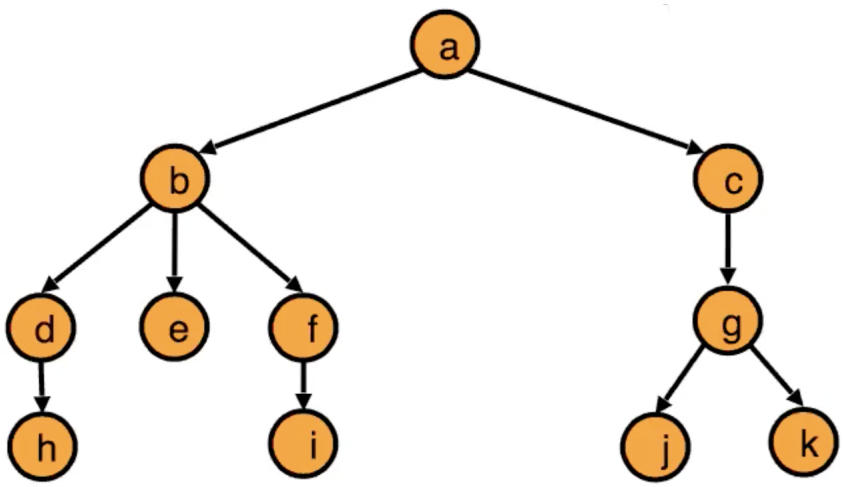
Definition. Infrastructure graph of directed graph $G$is a undirected graph with the same vertice in G and same edges in G but with no direction (e.g the graph 1.1 costiture example for graph1.2).

Definition. Source is a vertice which no edge point to it.

Definition. Directed tree is a directed graph with no cycles (in it's Infrastructure graph) and has only one sourced which is recalled by **root**.

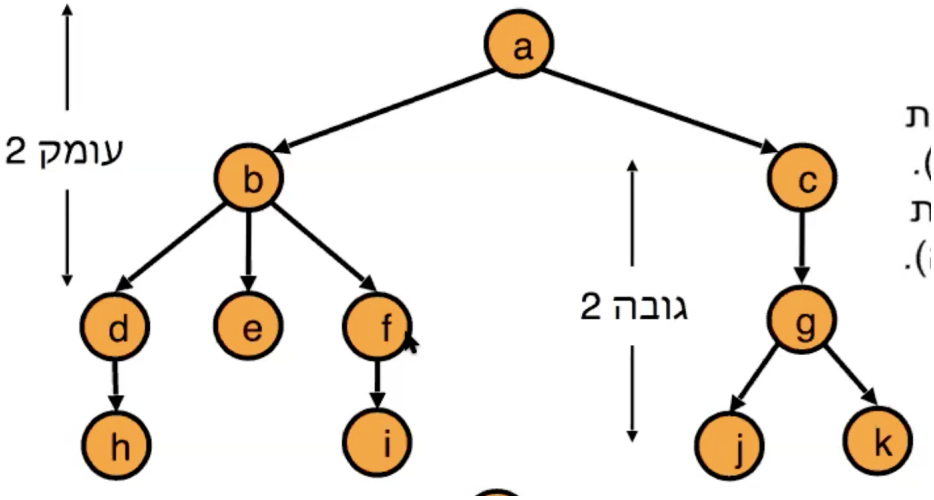
Definition. we are going to give some definitons:

- Vertice v **son** of u if there is edge point from u to v.
- Vertice u is **parent** of $v$if v is son of u.
- Vertice v is **lineal descendant** of u if exist a dircted path from u to v.
- Vertice u is **previous parent** of v if v is **lineal descendant** of u.
- **Subtree of G with a root v** is a directed tree in which it's vertices are $v$and all lineal descendant of v,and it's edges are the edges which connect thus vertices with G.
- **Degree of** v is number of sons of v.
- **Leaf** is a vertice with no sons.
- **Interior vertice** is a vertice which is not **leaf.**
- **Depth of vertice** v is a numer of edges from the root of the tree to v,
- **Heigh of vertice** v is a number of edges from v to it's farthest lineal descendant of v.
- **Tree Height** is a the height of it's root.



In the following graph we can see that:

- f is son of b.
- b is parent of e.
- g is a **lineal descendant** of a.
- b is previous parent of h.
- Subtree which is root g contain 3 vertices and 2edges.
- degree of $a$is 2.
- h is leaf.

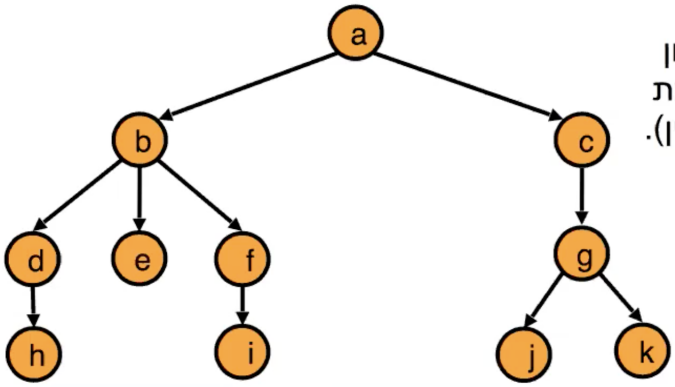


Remark. We can see in the following graph that the height of the tree is 3 Since the height of it's rows is number of edges from it's farthest lineal descendant which is $h \vee i \vee j \vee k$ (leafs of the tree).

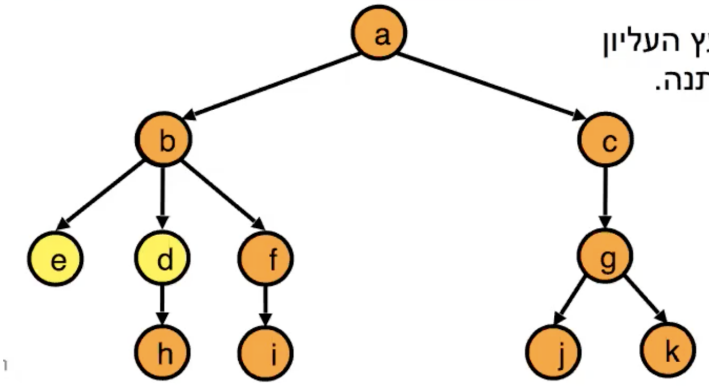
Remark. Sometimes we are going to omit the arrows from the graph since it's conventioanal that the directions in the edges are directed to down. and we are going to say also tree instead of directed tree for convenience considerations.

Definition. Ordered tree is is a deirected tree in which sons of every vertice are ordered from left to right.

We can look at the following ordered tree.



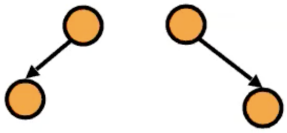
Now we can look at this tree which is different from the tree upove since the order of the sons in vertice b changed.



Remark. this order graph are useful we will say whyu it's important in case we want to search in graph or walk in graph.

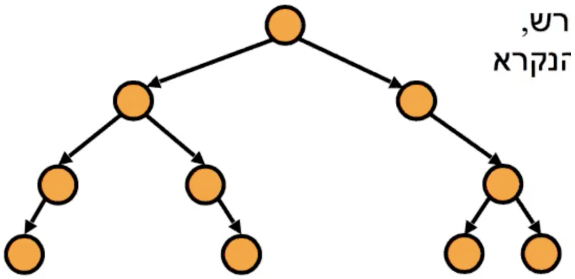
Binary Trees.

Definition. Binary tree is a tree in which for every vertice which is not leaf there is left son or right son.

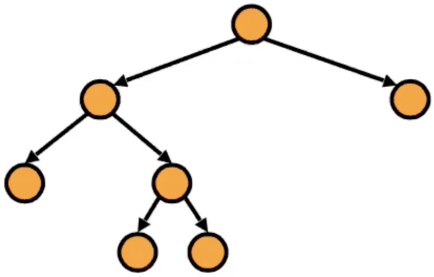


Definition. (recursive definition).
binary tree is a structure which is:

(1) empty (with no vertices).
(2) or built from three parts: vertice which is called root, binary tree which called by left subtree, and binary tree which called right subree.

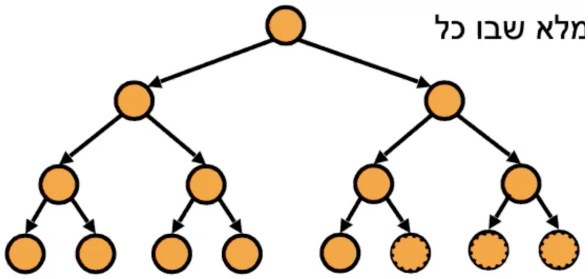


Definition. Binary fill tree: is a tree in which every interior vertice has two sons.



Definition. complete binary tree: is a binary tree in which all the leafs have the same depth.

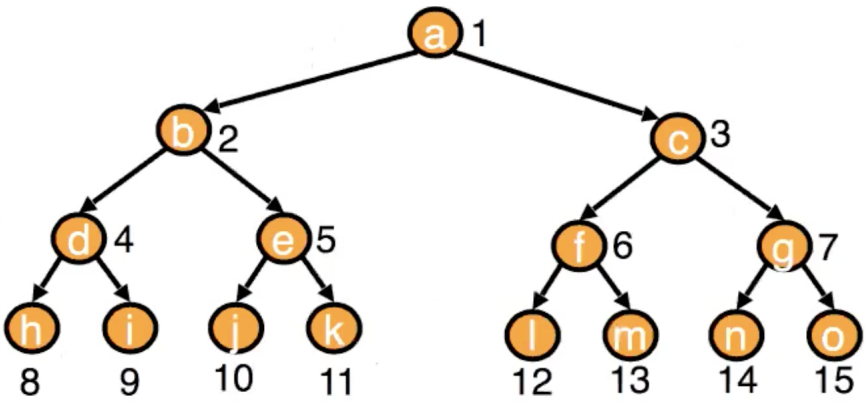
Definition. binary tree almost complete: is a full binary tree in which some leafs were deleted in the right side.



properties of a full binary tree with n vertives, L leafs, height h:

(1) number of vertices in depth i: is $n_i = 2^i$.
(2) number of leafs: $L = n_h = 2^h$.
(3) number of vertices $n = \sum_{i=1}^{h} n_i = \sum_{i=1}^{h} 2^i = 2^{h-1} - 1$.
(4) height is $h = log_2(n+1) - 1$.
(5) number of interior vertices are $n - L = 2^{h+1} - 1 - 2^h = 2^h(2-1) - 1 = 2^h - 1 = L - 1$.

representation in array of full binary trees.

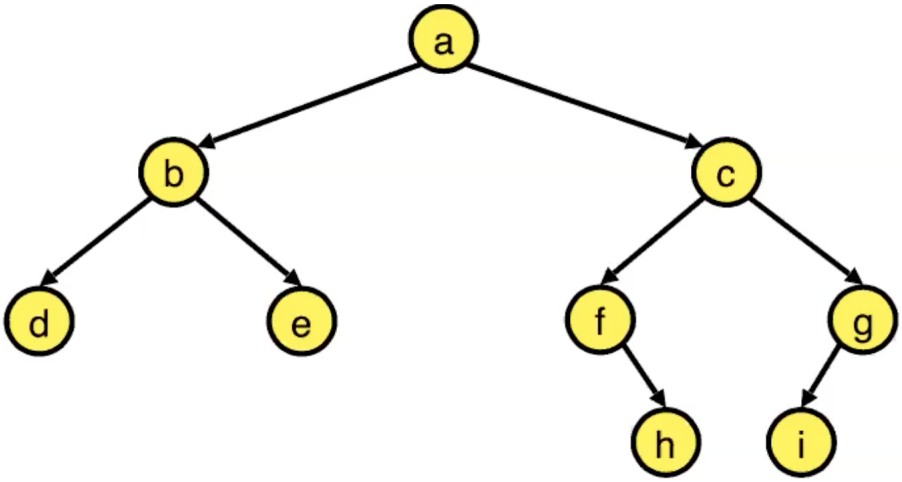


We can see the following:

$$\left\{ \begin{array}{cc} left-son-i, & 2i \\ right-son-i, & 2i+1 \\ parrnt-i, & \lfloor \frac{i}{2} \rfloor \end{array} \right\}$$

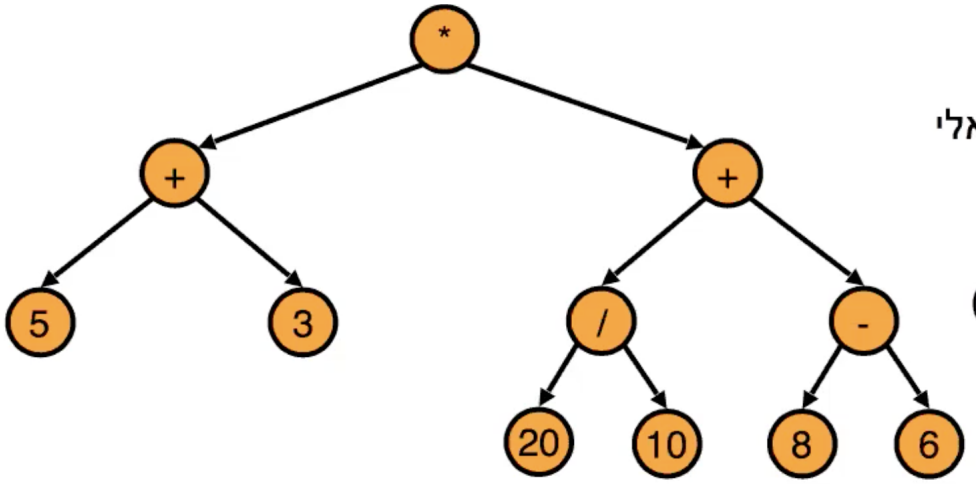
Remark. that why ordered full tree are important.

Tranversal in binary trees.



- Preorder.
  - visit the root.
  - visit the left subtree.
  - visit right subtree.
  - output: a b d e c f h g i.
- Postorder
  - visit the left subtree.
  - visit the right subtree.
  - visit the root.
  - output: d e b h f i g c a.
- inorder
  - visit the left subtree.
  - visit the root.
  - visit the right subtree.
  - output: d b e a f h c i g.

We can look at the transversal as a arithmetics term as descirbed in the following picture:



As we saw before inorder will give us the following formula:

$$((5+3)*((20/10)+(8-6)))-infix$$

As we saw before postorder will give us the following formula:

$$5\ 3{+}20\ 10/86{-}{+}-\ *-postfix$$

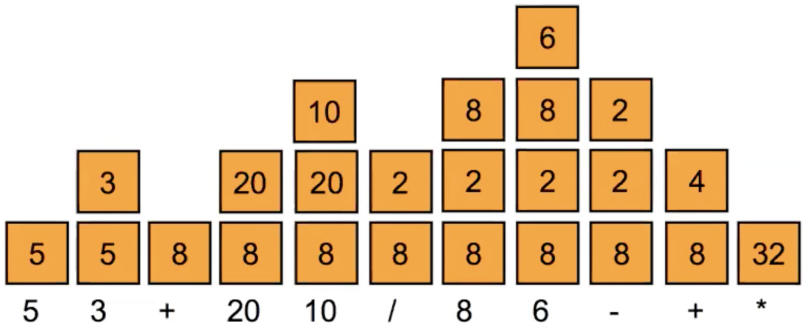
printing the term:

- if it's leaf
  - then print.
- otherwise
  - print a left bracket.
  - walk in the left subtree.
  - print root.
  - walk in the right subtree.

– print a right bracket.

Algorithm to calculate postfix term.

- start with empty stack.
- pass on the term from left to right.
- if the next element is operand - insert it to stack.
- if it's operation - operate the operation on the two elements in the head of the stack and insert the result to the stack.

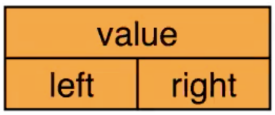


We can get the same result in infix which is 32 by doing the following using stack, so first we see number value then we insert it to stack and if we see a operation then look at latest number values and operate the opearation on them so first we have 5 and 3 and we insert it to the stack then we see the operation + so we operate it on the last numbers and get $5 + 3 = 8$ then we see other number value which is 20 so we insert it to the stack then we see the number 10 after that we see the operation / notice that the last number values which were inserted to the stack are $20, 10$ so we opearte the operation on them and get $20/10 = 2$ so currently we haave in the stack $8, 2$ then we see 8 number value and insert it, after that we see 6 then we have − operation again, we substract the last two number were inserted to our stacak and get $8, 2, 2$ in the stack then we see the opeation + and last two number were in the stack are $2, 2$ then we have $8, 4$ in the stack, finally we see $*$ operation and the last two numer are $8, 4$ so we multiply them and get $8 * 4 = 32$ which is the wanted result.

postorder implementation.

- Stucture of the vertice.

```
typedef struct node {
        int value;
        struct node *left, *right;
} NODE;
```



- Postorder tranversal implementation.

```
void postorder (NODE *T){
    if (T = = NULL) return;
    postorder( T → left);      /*1*/
    postorder( T → right);     /*2*/
    "visit"(T);                /*3*/
}
```

Remark. notice that by changing row 2 with 2 we get the tranversal implementation of inorder.

Exercise. try to change the code so it can calculate the result of the arithmetics term.

Solution. we can look at the following.

```
void postorder (NODE *T){
    if ("T is a leaf") return(T.value);
    val1 = postorder( T → left);                    /*1*/
    val2 = postorder( T → right);                   /*2*/
    return ("val1 operation(T.value) val2");        /*3*/
}
```

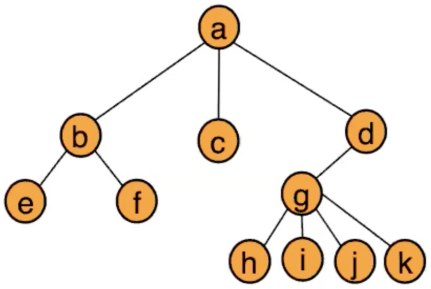
Exercise. try to write the code without using of recusrion. (using stack).

recusrsive function which calculate the height of tree (example for postoreder).

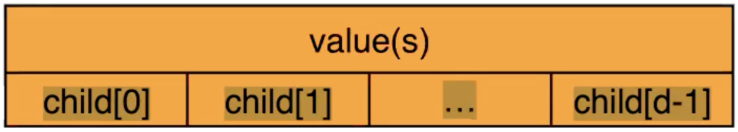
```
int height (NODE *T){
        int L,R;
        if (T = = NULL)  return  -1;
        else {
                L = height(T → left);
                R = height(T → right);
                return 1 + max(L,R) ;
        }
}
```

REPRESENTATION OF A VERTICE WITH BOUNEDED NUMBER DEGREE (I.E DEGREE OF VERTICE $\leq d$) (BINARY TREE REPRESNETATION).

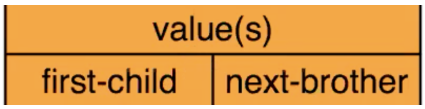
We can represent a pointer of a vertice by allocating two pointer for each vertice, first pointer point to the next brother to it and the second pointer point to it's first child. as the following,



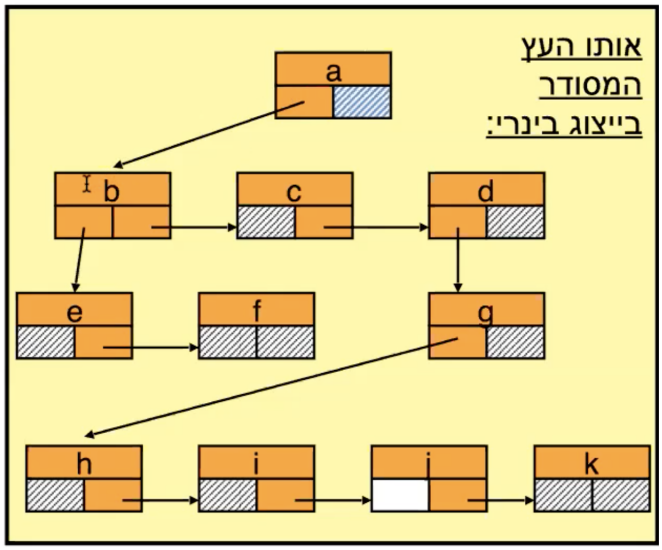
1.1.1.



1.1.2.



1.2.3.



We can see that E.g vertice a has pointer to it's first child since it's a root and it has no brothers, but vertice b has point to it's first child e and next brother which is c and e has point to it's brother (f) this method is not effective namely, in tranversal postorder since we have no point directly from $a$to d and we need to go by $a \to b \to c \to d$ which is not effective at all since we pass by unwanted ertice more than it really needed. hence, this represnetation is rarely used.

pros and cons of the representation.

- Advantage:

  - Before we required to allocated a lot of pointer in one verice to each son as in 1.1.1 but in the following representation we allocate only two pointer which is one to first child or one to the brother or both.
- Disadvantage:
  - We are required to do more long tranversal in the tree in order to reach specific node.
  - depth of the tree is bigger by d than before Since depth is numer of edges from the tree to root to the wanted vertice.

  So what is the realtion between the height of the source tree to the binary tree height?

Solution. NOTICE THAT $h_{new} \leq d \cdot h_{old}$ when d is the the maximal number of sons in the source tree.

Dictioanry data structure and implementation using tree.

Definition. Dicatiationry is collection of pairs in type (key,value) collection of possible keys denoted by U.

Dictionay support the following operations.

- $create()$ - return a new empty dictionary.
- $insert(D, x, info)$ - add the the collection of D the pair $(x, info)$.
- $delete(D, x)$ - delete from the collection of D the pair in which x it's key.
- $find(D, x)$ - if no such pair with a key x, return null. otherwise, retun the pair $(x, info)$ which were founded in the collection.

rules:

- x in the set of the keys U.
- every x appear at most one time in the dictionary.
- possible keys: natural numbers.

uses of dictionary:

- dictionarry: key=word, information: the word definition.
- Facebook, instgram, etc: key = username, information = all the user information.
- email: key = username, information = all the mails of the username.
- movies website: key = name of movie. information = movie.

Remark. we can notice that:

- In general the key is little, and information are big.
- it's trivial to keep in the dictionary the key and pointer to the place in which information were allocated.

other opearations when order is defined on U (E.g when the keys are numbers).

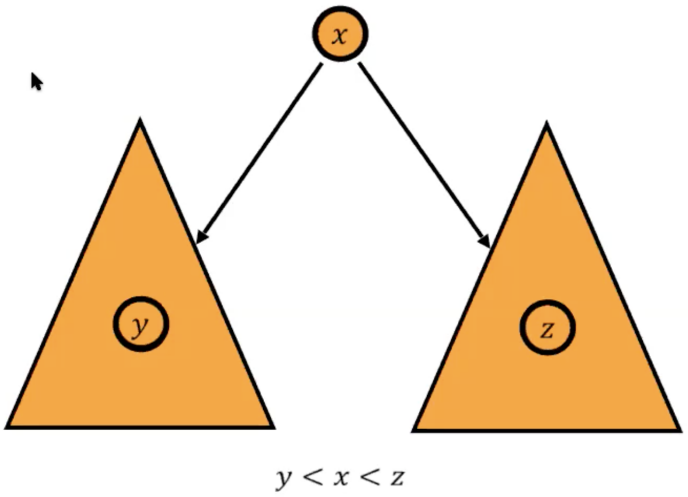
- $min(D)$ - return the minimal key in D,
- $next(D, x)$ - return the pointer to the element in the dictionary D which has the minimal key which is also bigger than x.

Proposition. *executing all the operations in time $O(logn)$ (in the worst case) when n is number of the keys which exist in the dictionary while running the operation.*

Searching trees.

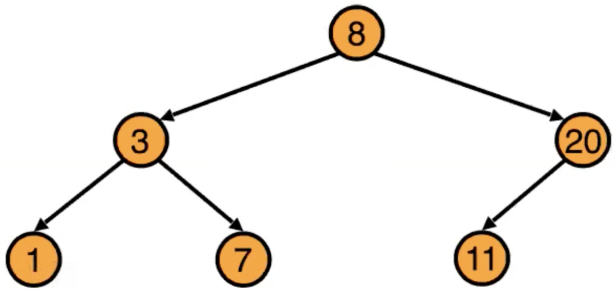
Definition. searching tree is a family of implementation to the data structure "dictionary" when order is defined in the set of keys U.

We will used directed binary trees. in each vertice we will store a key and a pointer to the information. we will preserve the following rule: for each vertice with a key x, all the vertices in the left subtree are smaller than x and all the keys in right subtree are bigger than x. as in the photo:



searching algorithm $find(T, x)$.

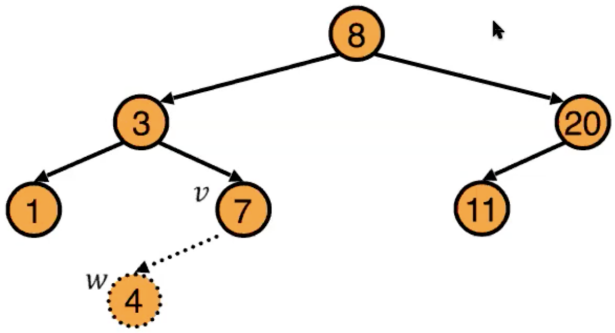
- if T empty, return that x is not in the tree.
- let y the value in the root.
- if $x = y$ return a pointer to the vertice which contain x.
- if $x < y$, complete the seaching in the left subrtree.
- otherwise, complete searching in the right subtree.



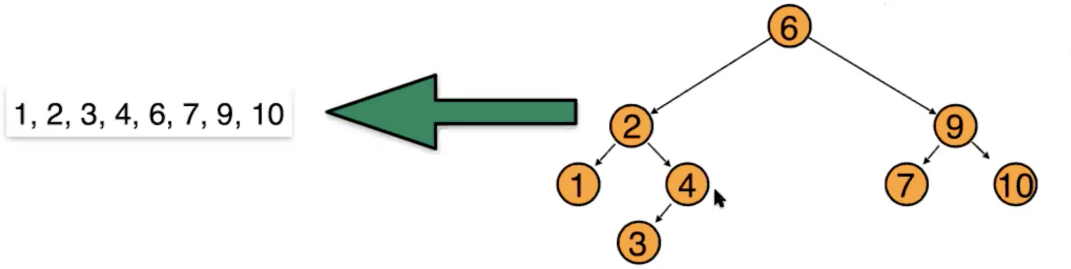
insertion algorithm $insert(T, x, info)$..

- find x in the searching tree T.
- if x exist in T stop and retrun suceess.
- let v the last vertice in the searching path of x and let y be the key exist in v.

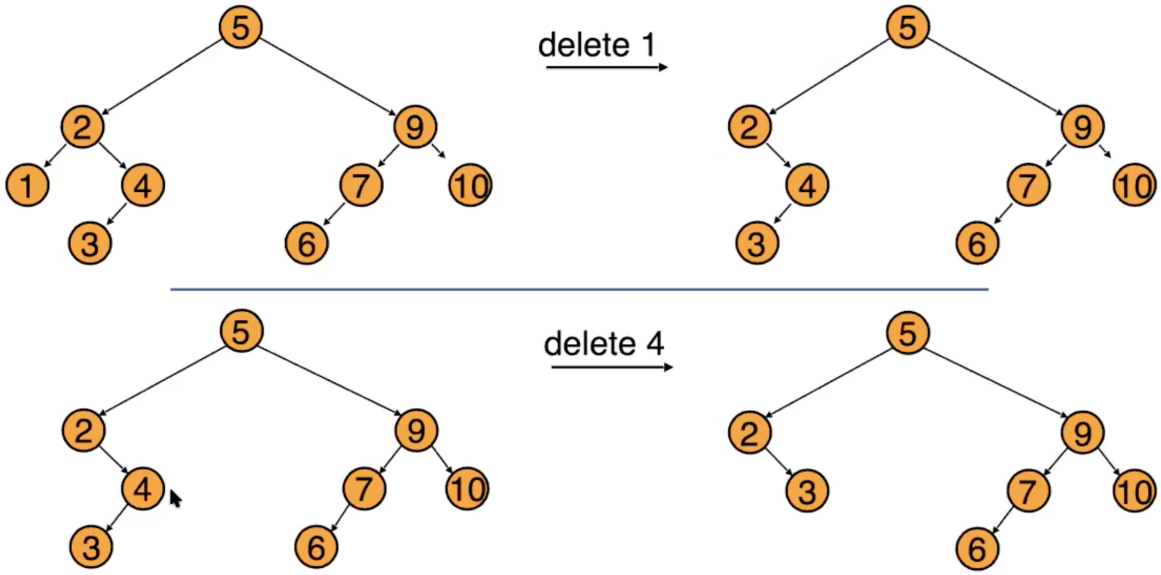
- if $x < y$, add vertice w with key x and data $info$ as a left son of v.
- if $x > y$, add vertice w with key x and data $info$ as a right son of v.



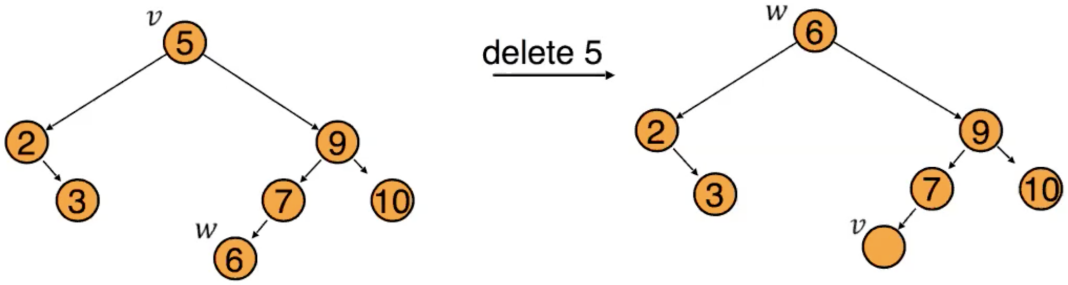
Remark. notice theat if the tree is a ordered tree as a searching tree then inorder tranversal will return the elements order by threre order as in the described photo.



deleting algorithm $Delete(x)$: *let* x *be the* v *be the vertice which are we asked to delete in the wanted tree.* 2.1



2.2



(1) if v is a leaf then it's easy to delete by deleting the pointer to it as (e.g deleting key 1 in photo 2.1).
(2) if v there is one son then we point it's father to for it son (e.g deleting key 4 in photo 2.1).
(3) otherwise we find the sequential key of x which is bigger than $x$and smaller than the other values, therefore, for finding it we got to the right subtree and walk in the left path then we change it with that key E.g in 2.2 we go to the right subtree beacuse we know that we are bigger than x then we go to the left path in order to find the sequential to it which is less than the other keys in the tree so we see in this case that it's 6 then we write it instead of x and we get rid of v using step mentioned previously method 1or 2 in this case $v$is a leaf so we use 1otherwise, we use 2. The reason for that is beacuse v has at most 1 son. so we continue by step 1 or 2.

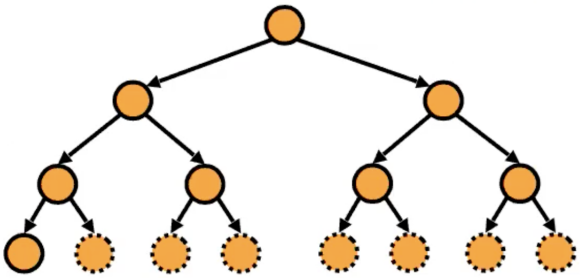
How tree will look as a sequence of insertion?

- if we insert $1, 2, 3, 4, 5, 6, 7$. (worst case -lanyard).
- if we insert $2, 4, 6, , 1, 3, 5, 7$.

time of searching/insertion/deleting is linear in tree height.

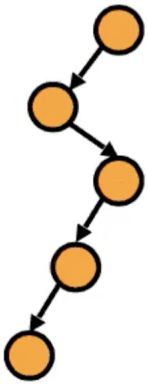
What is the tree height?

- Good case.
  - almost complete tree with height h :
    - $n \geq 2^h \Rightarrow log_2 n \geq h$.



  -

- Worst case.
  - "Lanyard" - tree which look like a linear list.i.e $h = n - 1$



So what is the expectation of height?

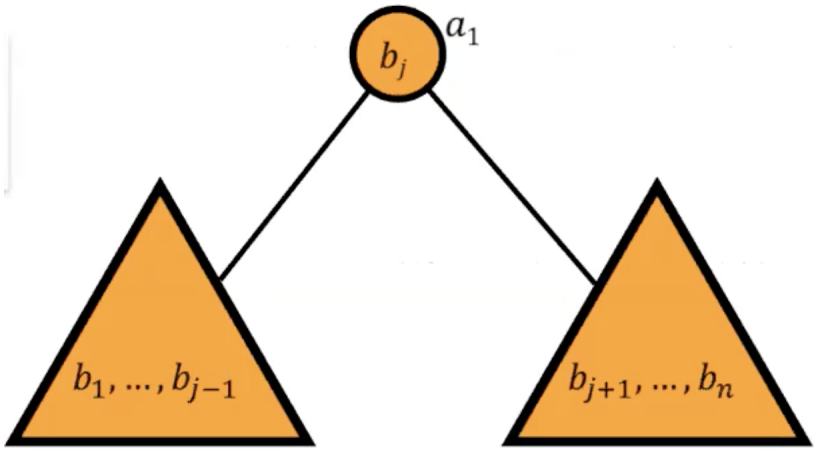
Solution. $O(nlogn)$. we will show.

- we don't know that much about the expectation height of tree in case there is a lot of operation instert and delete.
- we can talk about the heigh expectation if there is only insert. i.e we build the tree from inserts and we look at the height.
- as remembered order of inserting the elements determine the height.
- number of the optional sequence for inserting n elements $n!$.
- for every permutation p we build and new tree height which we will denote by $h(p)$.
- the expectation of height is:

$$E[h] = \frac{1}{n!} \cdot \sum_{p \in perm(n)} h(p)$$

- We can see that the expectation of height is in the set $O(logn)$ - i.e the expectations time of executing the opeartion is $O(logn)$.

expectation time of building searching tree. we will calculate the time of building random tree which we get by inserting random permutation $a_1, ..., a_n$ to a empty tree. we assume that the elements order is $b_1 < b_2 < .... < b_n$.



Remark. Some remarks for the insertion algorithm.

- first element we insert still forever our tree root.
- inserting time is equivalence (up to multiplying by constant) to the number of comparing we do while inserting.

So we will se the recursice formula $T(n)$ which represent the expecation of number of required compares to build tree with n vertices.

$$T(n) = \frac{1}{n} \cdot \sum_{j=1}^{n} [(n-1) + T(j-1) + T(n-j)] \qquad (T(0) = 0)$$

תוחלת הזמן הדרוש לבנות את העץ כש-b_j ראשון.

(n-1) איברים מושווים ל-b_j בשורש.

הזמן הדרוש לבנות את שני העצים מתחת ל-b_j.

Remark. we remind that expectation is probabity to choose permutation multiplying it's value height and we sum it some we have $n!$ different order and each one has height of $h(p)$ when $p \in perm(n)$ so we can look at the sum which is $E'[h] = \frac{1}{n!} \cdot \sum_{p \in perm(n)} h(p)$ thats how we got the formula of height expectations of the searching binary tree, we will show a proof that the expactation is going to be

our avarage height $log(n)$ from all the possible orders i.e expectation time of executing a opeartion is $O(logn)$ but we are going to represent that more easier proof which is expectation time of binary search tree which is going to be $O(nlogn)$.

intituirion of the recursive formula above. we saw that our recurstive formula is:

$$T(n) = \frac{1}{n} \cdot \sum_{j=1}^{n} [(n-1) + T(j-1) + T(n-j), T(0) = 0$$

So we can see two different case, assuming that the most smallest element was places in the root hence, in this case in the left subtree will be zero elements, notice that in j denote the insertion position of the element E.g $j = 1$ mean that first element were inserted is b_j and it's going to be our tree root , Moreover, we have $T(n-1)$ time for constructing the right subtree. So if we plug in $j = 1$ in the formula we get that:

$$(n-1) + T(0) + T(n-1)$$

as we said we have $T(0) = 0$ time to build the left subtree and $T(n-1)$ to build the right subtree, now notice that each one of the $n-1$ we comared it to the root in order ro check if they are in the left or right subtree and it takes $n-1$ compare operation so that why we add $n+1$ to our recursive formula. in case $j = 2$ we will have 2 vertices in the left subtree and $n-2$ vertices $n-2$ in the right subtree i.e

$$(n-1) + T(2) + T(n-2)$$

hence, we get the following recursive formula. now we will proof that this recursive formula is $O(nlogn)$ which represent the time of constructing the searching binary tree.

Proof. the formula we got is:

$$T(n) = \frac{1}{n} \cdot \sum_{j=1}^{n} [(n-1) + T(j-1) + T(n-j) =_* n - 1 + \frac{2}{n} \sum_{j=0}^{n-1} T(j)$$

When in $*$ we use symmetry property since there is elements which appear in the sum twice.

Now we multiply both side by n and get that:

1.

$$nT(n) = n(n-1) + 2 \sum_{j=0}^{n-1} T(j)$$

Identically if we plug-in $n-1$ in the formula we get that:

2.

$$(n-1)T(n-1) = (n-1)(n-2) + 2 \sum_{j=0}^{n-2} T(j)$$

Substracting formula 1 from 2 give us:

$$nT(n) - (n-1)T(n-1) = 2(n-1) + 2T(n-1)$$

$$\Downarrow$$

$$nT(n) = 2n - 2 + (n+1)T(n-1)$$

deviding both side by $n \cdot (n+1)$:

$$\frac{T(n)}{n+1} = \frac{2}{n+1} - \frac{2}{n(n+1)} + \frac{T(n-1)}{n} <_{*} \frac{2}{n+1} + \frac{T(n-1)}{n}$$

$$<_{**} \frac{2}{n+1} + \frac{2}{n} + \frac{T(n-2)}{n-1} < ... < 2\sum_{j=1}^{n} \frac{1}{j+1} + \frac{T(0)}{1}$$

Remark. in $**$ we apply $*$ for $T(n-1)$. Moreoverm H_n is the hermonic sum of n and it satsify that $H_n = \sum_{j=1}^{n} \frac{1}{j} = O(logn)$.

In total we got that

$$<_{**} \frac{2}{n+1} + \frac{2}{n} + \frac{T(n-2)}{n-1} < ... < 2\sum_{j=1}^{n} \frac{1}{j+1} + \frac{T(0)}{1} = 2(H_{n+1} - 1) + 0 = O(log(n+1)) = O(logn)$$

hence,

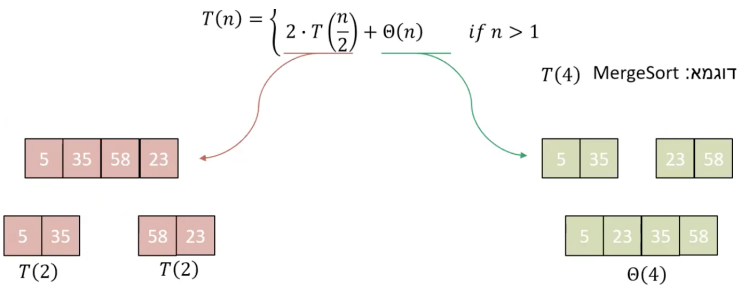
$$\frac{T(n)}{n+1} = O(logn) \Rightarrow T(n) = O(nlogn).$$

□

Recursive equations.

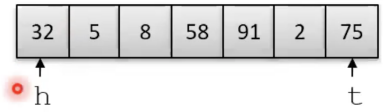
Definition. recursive is a equation or inequality, which define function by function values on small argoments E.g

$$T(n) = \left\{ \begin{array}{ll} \Theta(1), & if\ n = 1 \\ 2 \cdot T(\frac{n}{2}) + \Theta(n), & if\ n > 1 \end{array} \right\}$$

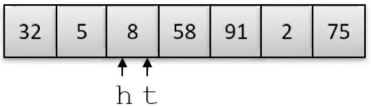


Example. We look at the following algorithm which find the maximal element in a not sorted array A, when h denote the beggining searching index and t is the end searching index.

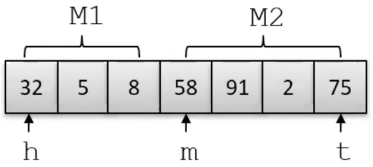
```
Max(h,t)
     if (h == t) return A[t];
     else {
            m  = (h + t) / 2;
            M1 = Max(h,m-1);
            M2 = Max(m,t);
            return (M1 > M2) ? M1 : M2;
     }
```




```
     if (h == t) return A[t];
     else {
            m  = (h + t) / 2;
            M1 = Max(h,m-1);
            M2 = Max(m,t);
            return (M1 > M2) ? M1 : M2;
     }
```



```
else {
        m  = (h + t) / 2;
        M1 = Max(h,m-1);
        M2 = Max(m,t);
        return (M1 > M2) ? M1 : M2;
}
```



max algorithm complexity. We write recursice equation $T(n)$ which describe algorithm running time, when n denote the array length.

$$T(n) = \left\{ \begin{array}{ll} c_1 & if\ n = 1 \\ 2 \cdot T(\frac{n}{2}) + c_2, & if\ n > 1 \end{array} \right\}$$

Remark. c_2 is the compare bewteen M_2, M_1 time and returnin the biggest value of them.

> How could we solve the following recursion and deduce that it's $O(n)$?

methods to solve recustion.

0.1. **The inductive method.**

- showing asymptotics notaitons (O, Ω, Θ) using induction.
  - The method is used for case in which we want to insure conjecture i,e in the previous example we know that it's $O(n)$ so we want to insure that it's indeed.
  - Notations asymptotics constants will be figured out (c, n_0) while the proof.

Example. Look at :

$$T(n) = \left\{ \begin{array}{ll} 1 & if\ n = 1 \\ 2 \cdot T\left\lfloor \frac{n}{2} \right\rfloor + n, & if\ n > 1 \end{array} \right\}$$

We will try to show that $T(n) = O(nlogn)$.

Proof. We wish to find $c, n_0 > 0$ S.T $\forall n \geq n_0$ satsified that $T(n) \leq c \cdot nlog_2 n$.
We will show the following lema by induction.

Lemma. *$\forall n \geq n_0$ satisfied that $T(n) \leq c \cdot nlog_2 n$.*

Proof. Assuming that the lema is true for $k = \left\lfloor \frac{n}{2} \right\rfloor$. i.e: $T(k) \leq c \cdot klog_2 k$.
First we will show the induction step:
Step: we will operate the induction assumption in the recursive definition:

$$T(n) = 2 \cdot T(\left\lfloor \frac{n}{2} \right\rfloor) + n \leq 2 \cdot (c \cdot \left\lfloor \frac{n}{2} \right\rfloor \cdot log_2 \left\lfloor \frac{n}{2} \right\rfloor) + n$$

$$\leq 2 \cdot (c \cdot \frac{n}{2} \cdot log_2 \frac{n}{2}) + n = c \cdot n(log_2 n - log_2 2) + n$$

$$= c \cdot n \cdot (log_2 n - 1) + n = c \cdot n log_2 n - cn + n \leq_* c \cdot n \cdot log_2 n$$

Remark. notice that $*$ is true by choosing $c \geq 1$. which is fine because we need to find specefic c.

□

□

Remark. Notice that here we didn't assume the induction step from $n-1$ and show to n.

why regular induction work? Notice that:

Base: for $n = 1$.

Step: $n - 1 \Rightarrow n$.

We want to know if the induction work for $n = 3$. so how the induction help?

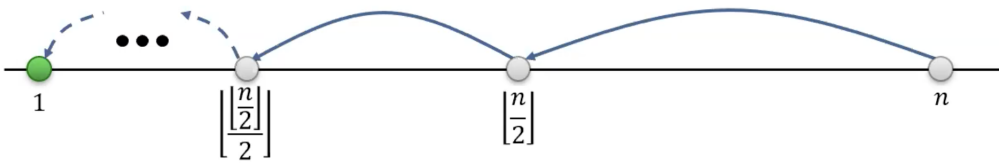
Proof sequence:

- The lema true for $n = 3$ if it's true for $n = 2$.
- The lema true for $n = 2$ if it's true for $n = 1$.
- The lema is true for $n = 1$ (base).

In tha lema above we asasumed that the following is true for $\lfloor \frac{n}{2} \rfloor$ and we show correctness for n i.e the induction step $\lfloor \frac{n}{2} \rfloor \Rightarrow n$. assuming that we succeed to show the base $n = 1$ satisfied, why it's enough to prove the claim?

- We wish to show that the claim satisfied for $n \geq 1$ by base 1 and step we already showed.
- by the induction step, the claim satisfied for n if it's satisfied for $\lfloor \frac{n}{2} \rfloor$.
- The claim satisfied for $\lfloor \frac{n}{2} \rfloor$ if it's satisfied for $\left\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2} \right\rfloor$.
- and etc..

hence, the source claim is true since for $n \geq 1$ we can build proof sequence as following which reach the base case 1.



back to the example.

$$T(n) = \left\{ \begin{array}{ll} 1 & if\ n = 1 \\ 2 \cdot T\left\lfloor \frac{n}{2} \right\rfloor + n, & if\ n > 1 \end{array} \right\}$$

Induction Base: we try wo show that claim for $n = 1$: $T(n) \leq c \cdot nlog_2 n$

$$1 = T(1) \nleq c \cdot 1log_2 1 = 0$$

in order to void this problem we remember that the analyze we are executing is asymptotic. i.e we need to show $n_0 > 0$ in which from it the claim start to be satisfied.

first try, Taking as the induction base $n = 2$, for $c \geq 2$ satisfied:

$$4 = T(2) \leq c \cdot 2log_2 2 = c \cdot 2 = 1$$

So it's enough? NO!!

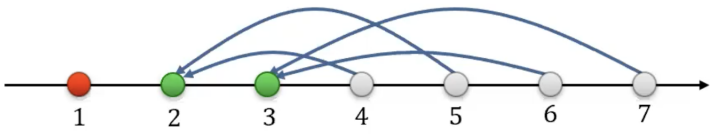
- We can't show proof sequence for $T(3)$ Since $T(\left\lfloor \frac{3}{2} \right\rfloor) = T(1)$ and we know that for $n = 1$ it's not true.

other try. Taking the induction base as $n = 2$ and $n = 3$ and for $c \geq 2$ we have that:

$$4 = T(2) \leq c \cdot 2 \cdot log_2 2$$

$$5 = T(3) \leq c \cdot 3 \cdot log_3 2$$

Now, we can use the inductive proof in order to show the lema for $n \geq 2$. hence, if we take $n_0 = 2, c = 2$ the claim satisfied as we see in the photo.



Remark. Notice that we show that exist $c, n_0 > 0$ in which the following lema satisfied

Lemma. $\forall n \geq n_0$ *satisfied* $T(n) \leq c \cdot nlog_2 n$.

hence, in total we showed that the claim is true for the n_0 *we chose which in our case was* $n_0 = 2$ *and we show that it's true for* $c \geq 1$ *and we took* $c = 2$ *so it's satisfy also.*

0.2. **The itiration method.**

- We don't know in advance the solution.
- We open the recursive term as sum of element which are dependent in n and in the intial consition.

Example. We can look at:

$$T(n) = n + 3 \cdot T(\left\lfloor \frac{n}{4} \right\rfloor) = n + 3(\left\lfloor \frac{n}{4} \right\rfloor + 3 \cdot T(\left\lfloor \frac{n}{4} \right\rfloor))$$

$$= n + 3 \cdot \left\lfloor \frac{n}{4} \right\rfloor + 9 \cdot \left\lfloor \frac{n}{16} \right\rfloor + 27 \cdot T(\left\lfloor \frac{n}{64} \right\rfloor)$$

What is the number of itiration we need to execute in order to reach the intial condition (i.e $T(1)$) ?

$$\lfloor \frac{n}{4^i} \rfloor = 1 \Rightarrow i = log_4 n$$

$$T(n) = n + 3 \cdot T(\lfloor \frac{n}{4} \rfloor) = n + 3(\lfloor \frac{n}{4} \rfloor + 3 \cdot T(\lfloor \frac{n}{4} \rfloor))$$

$$= n + 3 \cdot \lfloor \frac{n}{4} \rfloor + 9 \cdot \lfloor \frac{n}{16} \rfloor + 27 \cdot T(\lfloor \frac{n}{64} \rfloor)$$

$$\leq n + \frac{3}{4} = n + \frac{3}{4}n + \frac{9}{16}n + ... + 3^i \cdot T(\lfloor \frac{n}{4^i} \rfloor)$$

$$\leq n \cdot \sum_{i=0}^{\infty} (\frac{3}{4})^i + 3^{log_4 n} \cdot \Theta(1) \leq n \cdot \frac{1}{1 - \frac{3}{4}} + 3^{log_4 n} \cdot \Theta(1) = 4n + 3^{log_4 n} \cdot \Theta(1)$$

Remark. Notice that:

$$3^{log_4 n} = 3^{\frac{log_3 n}{log_3 4}} = n^{\frac{1}{log_3 4}} = n^{\frac{1}{(\frac{log_4 4}{log_4 3})}} = n^{\frac{1}{(\frac{1}{log_4 3})}} = n^{log_4 3}$$

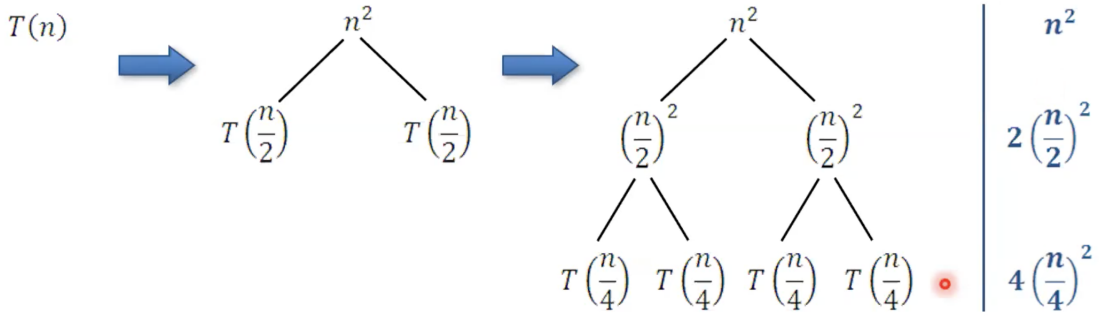
Now notice that $log_4 x$ is a continious increasing monotonic function in the interval $[1, \infty]$ hence, $\forall x \in \mathbb{R}$ we can look at the seqence $x = n$ since it's continious a heine craterion is satified under the condition hence $n^{log_4 3} \leq n^{kog_4 4} = n^1 = n$ so in total we got that:

$$4n + 3^{log_4 n} \cdot \Theta(1) \leq 5n \cdot \Theta(1) = O(n)$$

0.3. **Recursion trees.** A comfort way to represent a recusrive term when we open it, identical to the itiration method.

Example. Look at:

$$T(n) = 2 \cdot T(\frac{n}{2}) + n^2$$



we can see that the some of all the nodes in the graph is $T(n)$.

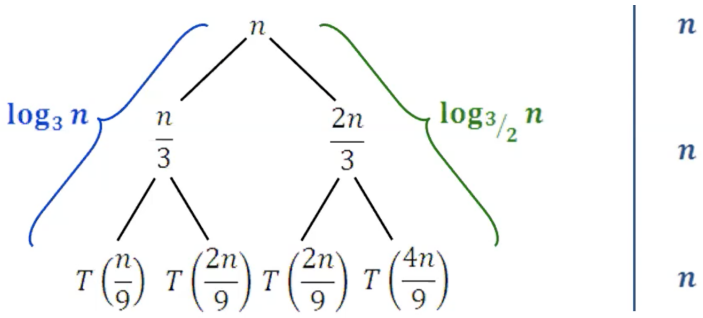
Remark. notice that the intiuition is just look at every itiration as a previous subtree in the beggining we put in our tree root n^2 and we connect it with two nodes each one is $T(\frac{n}{2})$ and we can do it again and again till we get maxiaml tree heigh which will be when we arrive the intial condition of the recusrive equation.

$$T(n) \leq n^2 + 2 \cdot (\frac{n}{2})^2 + 4 \cdot (\frac{n}{4})^2 + ... \leq n^2 \cdot \sum_{i=0}^{\infty} \frac{1}{2i} = 2n^2 \Rightarrow T(n) = O(n^2)$$

Example. Look at:

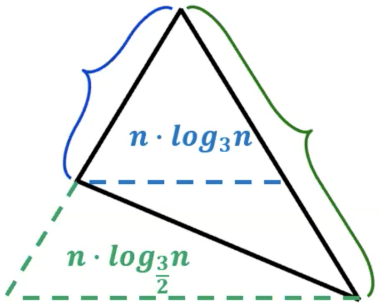
$$T(n) = n + T(\frac{n}{3}) + T(\frac{2n}{3})$$

The following recursion represent tree in this following form, the tree has right and left subtree, each one of thus subtrees has a different height.



We can calculate the height of each subtree.

notice that the shortest height is when $(\frac{n}{3^i}) = 1 \Rightarrow i = log_3 n$ Moreover, the longest height is $n(\frac{2}{3})^i = 1 \Rightarrow log_{\frac{3}{2}} n = i$ now we want to find good bounders we can describe it as triangle.



Notice the the sum of each line in the tree is n and we have minimal height of $n \cdot log_3 n$ hence, $n \cdot log_3 n \leq T(n)$ Moreover, the longest height is $nlog_{\frac{3}{2}} n$ and each line has n sum therefore, $n \cdot log_3 n \leq T(n) \leq nlog_{\frac{3}{2}} n$ so we got that $T(n) = \Theta(nlogn)$.

0.4. **The master method.**

- let $a \geq 1, b > 1$ constants.
- $f(n)$ function.
- $T(n)$ defined on $\mathbb{N}$ by the recustion $T(n) = a \cdot T(\frac{n}{b}) + f(n)$.

We can bound $T(n)$ asymptotc in the following way.

(1) if $f(n) = O(n^{log_b a - \epsilon})$ when $\epsilon > 0$ then $T(n) = \Theta(n^{log_b a})$.
(2) if $f(n) = \Theta(n^{log_b a})$ then $T(n) = \Theta(n^{log_b a} \cdot logn)$.
(3) if $f(n) = \Omega(n^{log_b a - \epsilon})$ when $\epsilon > 0$ Moreover, $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ for $c < 1$ and big enough n then $T(n) = \Theta(f(n))$.

Remark. The meaning of $\frac{n}{b}$ can also be $\lfloor \frac{n}{b} \rfloor \vee \lceil \frac{n}{b} \rceil$.

Exercise. Look at the following table.

$T(n) = a \cdot T(\frac{n}{b}) + f(n)$	a	b	$log_b a$	$f(n)$	Θ
$T(n) = 9 \cdot T(\frac{n}{3}) + n$	9	3	2	n	$(1).n^2$
$T(n) = T(\frac{2n}{3}) + 1$	1	$\frac{3}{2}$	0	1	$(2).logn$
$T(n) = 3 \cdot T(\frac{n}{4}) + nlogn$	3	4	$log_4 3$	$nlogn$	$(3).nlogn$
$T(n) = 1 \cdot T(\frac{n}{2}) + nlogn$	2	2	1	$nlogn$	

Notice that in the first example $T(n) = 9 \cdot T(\frac{n}{3}) + n$ we check the three conditions we start with the first one. we ask ourself if exist $\epsilon > 0$ S.T $n = O(n^{2-\epsilon})$ E.g $\epsilon = 1$ and the condition satsifed hence, $T(n) = \Theta(n^2)$ by 1. Now the second recusrsion $T(n) = T(\frac{2n}{3}) + 1$ we can see the f is $\Theta(1)$ therefore we check the second case i.e $f(n) = \Theta(n^{log_b a}) = \Theta(1)$ which is true therefore $T(n) = \Theta(n^{log_b a} \cdot logn) = \Theta((2)logn)$. Now the third recusrion $T(n) = 1 \cdot T(\frac{n}{2}) + nlogn$ notice that there is no $\epsilon > 0$ S.T $f(n) = nlogn = O(n^{log_4 3 - \epsilon})$ since $n^{log_4 3} < n$ so we are asking if $nlogn = O(n)$ which is not true. so we move to the next condition which is checking if $f(n) = \Theta(n^{log_4 3})$ which is also not true. now we move to the third condition we need to find $\epsilon > 0$ S.T $nlogn = \Omega(n^{log_4 3 + \epsilon})$ so now we check if there is $c < 0$ S.T $3 \cdot \frac{n}{4} log \frac{n}{4} \leq c \cdot nlogn$ we can simply take $c = \frac{3}{4} < 1$ and the third condition satisfied hence, $T(n) = \Theta(nlogn)$. now the fourth recuston which is $T(n) = 1 \cdot T(\frac{n}{2}) + nlogn$ we notice that 1 is not satisfied since $f(n) = nlogn \neq O(n^{1-\epsilon}), \forall \epsilon > 0$ Moreover, second condition not satisfied since, $f(n) = nlogn \neq \Theta(n^1)$ and the third condition also since, we can see that there is no $\epsilon > 0$ S.T $nlogn = \Omega(n^{1+\epsilon}) \iff logn = \Omega(n^{\epsilon})$ since we showed previously lema claim that $logn = o(n^{\epsilon})$. So what we do in this case? simply we check other methods instead.

Recursion - variable change.

Exercise. Find a asymptotic bound to the following recursive equation.

$$T(n) = \left\{ \begin{array}{ll} 1, & n = 1 \\ 2 \cdot T(\sqrt{n}) + (logn - 1), & n > 1 \end{array} \right\}$$

Step 1:
denote $m = log_2 n$ which is equivalence to say $n = 2^m$.

$$T(2^m) = 2 \cdot T(2^{\frac{m}{2}}) + m - 1$$

Step 2:
Define $S(m) = T(2^m)$.Notice that under the following definition we have that $S(\frac{m}{2}) = T(2^{\frac{m}{2}})$.

$$S(m) = 2 \cdot S(\frac{m}{2}) + m - 1$$

We saw previously that $S(m) = \Theta(mlogm)$ hence,

$$T(n) = T(2^m) = s(M) = \Theta(mlogm) = \Theta(logn \cdot loglogn)$$

Exercise. Find a asymptotic bound to the following recursive equation.

$$T(n) = \left\{ \begin{array}{ll} 1, & n = 1 \\ 2\sqrt{n} \cdot T(\sqrt{n}) + n(logn - 1), & n > 1 \end{array} \right\}$$

First we devide the equation by n.

$$\frac{T(n)}{n} = 2 \cdot \frac{T(\sqrt{n})}{\sqrt{n}} + logn - 1$$

Step 1:
We dentoe $U(n) = \frac{T(n)}{n}$. we notice that under the definition, $U(\sqrt{n}) = \frac{T(\sqrt{n})}{\sqrt{n}}$ hence,

$$U(n) = 2 \cdot U(\sqrt{n}) + logn - 1$$

We got the same recusrion in previous exercise, and we know that:

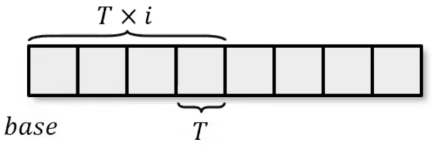
$$T(n) = n \cdot U(n) = n \cdot \Theta(logn \cdot loglogn) = \Theta(n \cdot logn \cdot loglogn)$$

Remark. it's trivial that $n \cdot \Theta(logn \cdot loglogn) = \Theta(n \cdot logn \cdot loglogn)$ if you still not convinced you can show it quick by finding a good constant $c, n_0 > 0$ which satisfy the definition and prove that it's true and legal to do it.

Arrays

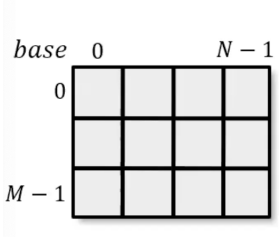
Definition. one dimenition array $A[N]$.

- Spacae complexity $O(N)$.
- The address of the i in the array is $base + i \cdot T$
- hence, the complexity time to access an i element in the array is $O(1)$.



Definition. Two dimentional array $A[m][n]$

- complexity time $O(mn)$.
- The Two dimentional array represneted in the memory as a one dimentional array, by rows.
- The address of the (i, j) element in the array is $base + (i \cdot N + j) \cdot T$.
- hence, the complexity time to access an i, j element in the Two dimentional array is $O(1)$.



Special matrices: diagonal matrices. A square marix M_{NXN} in which elements of every diagonal are identical elements E.g

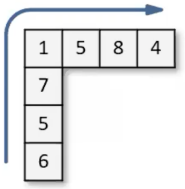
	0	1	2	3
0	1	5	8	4
1	7	1	5	8
2	5	7	1	5
3	6	5	7	1

We want to support the following operation:

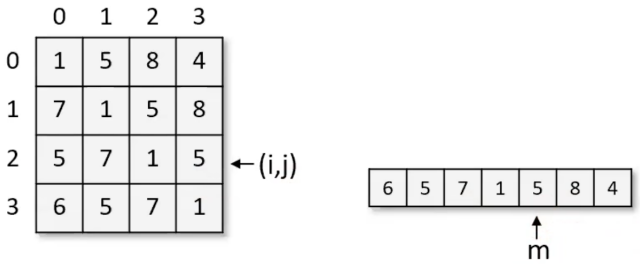
- $Get(i, j)$ - return the element in cell $M[i][j]$.
- $put(i, j, x)$ - store x in cell $M[i][j]$. we need to update all the diagonal.

	put	*get*	*space complexity*
Standard implementation	$O(N)$	$O(1)$	$O(N^2)$

Standard implementation (naive): By two dimentional matrix.
Modified implementation: We store onlt the first row and the first column.
1.1



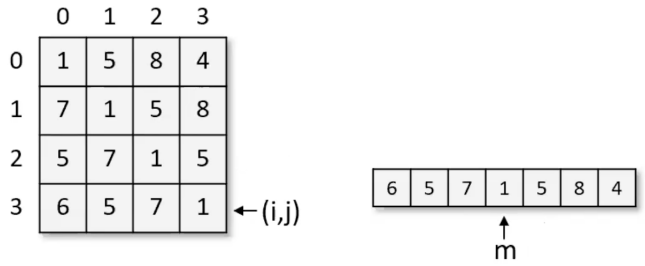
1.2



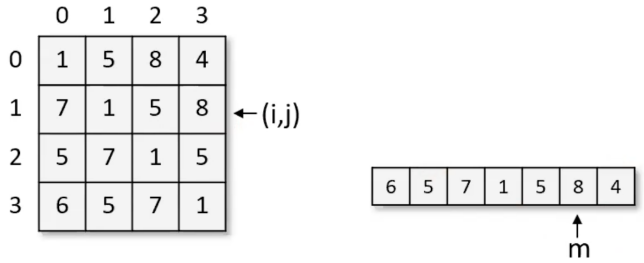
We can see the in every diagonal in the matrix are identical so it's enough to store first rwo and first column as described in the photo 1.2. now in the modified implementation E.g we look at the element (i, j) above in the photo 1.2 we see that it's point to 5 now it's adapt the element which m point to in(arrray union of first

column and row). So we can split our problem to cases in order to find the relation between index (i, j) to m.

- What happen if we increment i by 1i.e:



- What happen if we increment i by -1 i.e:



- What happen if we increment j by 1.
- What happen if we increment j by -1.
- What is m for $(0, 0)$? $m = N - 1 = 3$.
- General formula:
  - $m = N - 1 + j - i$.

  Remark. notice that E.g we are looking at $(2, 1)$ which is 7 in our case, we want a formula which give us 7 in the one dimentional array. so we can see that we have $m = N - 1 + (-1) = N - 2 = 2$ and it work for other all other values.

	put	*get*	*space complexity*
Modified implementation	$O(1)$	$O(1)$	$O(N)$

Remark. in the modified implementation we see that put reqiure $O(1)$ because we don't have to put the element x in all the diagonal it's enough to put it in one place in the one dimentional array Moreover, we improved the space complexity since the new array is contained of one row and one column each one of length N hence, $2N - 1$ which is $O(N)$ and it's way better than naive implementation which require $O(N^2)$

Remark. finding the general formula for m is not a easy thing. but we can start whith staff we know like checking the base $(0, 0)$ we see that it's in $N - 1$ in the union array. now we want to find formula wh/ich also cover other elements we can see that for example element 5 is in the place (i, j) in which $j = i + 1$ hence, $j - i = 1$ therefore, adding difference one always is a good way to make all the 5 at the third digonal spacial than other elements, for example element 7 exists in (i, j)

S.T $i = j + 1 \to j - i = -1$ and 6 exist in (i, j) S.T $i = j + 3 \to j - i = -3$ so we can do our array like that:

N-1-3 (6 at same diagonal)	$N-1-2$ (5 at same diagonal)	$N-1-1$ (7 at same diagonal)	..	..		

Special matrices: three diagonal matrices.

Definition. three digonal matrices is a squared matrix M_{NxN} in which all the elements are equal to constant except three longest diagonals.

Example. Look at the following matrix.

1	2	0	0	0
5	4	1	0	0
0	0	3	0	0
0	0	1	2	3
0	0	0	7	9

	put	*get*	*space complexity*
Standard implementation	$O(1)$	$O(1)$	$O(N^2)$

Is there better way?

Standard implementation: by a two dimentional matrix.

Modified implementation: We keep only the longest three diagonals in one array as the following.

אחר	0,1	1,2	2,3	3,4	0,0	1,1	2,2	3,3	4,4	1,0	2,1	3,2	4,3
0	2	1	0	3	1	4	3	2	9	5	0	1	7

In order to calculate the right cell, it's enough to understand in which diagonal we are.

- How we conclude that it's the middle diagonal?
  - $i == j$
- How we conclude that it's the up diagonal?
  - $j == i + 1$
- How we conclude that it's the down diagonal?
  - $j == i - 1$
- How we conclude that we are out of all diagonals?
  - else

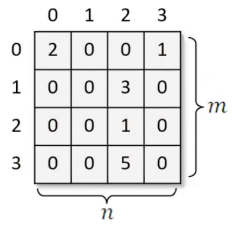
Example. we can look at index $(1, 4)$ it's not any of the first three options therefore, it's the else sitiuation i.e we have 0.

	put	*get*	*space complexity*
Modified implementation	$O(1)$	$O(1)$	$O(N)$

Special matrices: sparness matrices.

Definition. a mtrix in which almost all of the elements are equal to constant we know that r is bound on the number of the elements in which are not 0 and satisfied: $r << m \cdot n$.

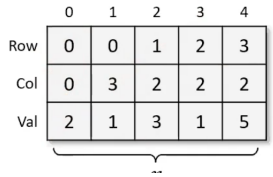
	put	*get*	*space complexity*	$Transpose$
Standard implementation	$O(1)$	$O(1)$	$O(mn)$	$O(mn)$



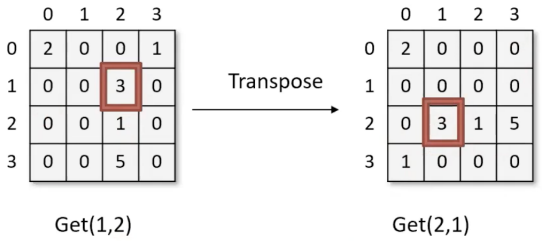
Is there better way?

Modified implementation: we keep only the elements which are not 0, in the following way:

- Every element x which are not 0 which exists in cell (i, j) represented by (i, j, x).
- Elements which are sorted in lexicographic order of (i, j). i.e, they are sorted first by i then by j.



Remark. in our modified implmentation we can use binary search for opeartion *get* since it's sorted in Lexicographic order therefore, $O(logr)$ time complexity. Moreover, for *put* operation we have that $O(logr)$ in order to seach for the place which we want to put our element, and $O(r)$ in order to move all the elements after we put it. therefore $O(logr) + O(r) = O(r)$. for the trnaspose operation we have that $O(rlogr)$ Since we want to sort in Lexicographic order again by column first and by row second. But notice that we can doit in $O(1)$. by inverting our input instead of sorting again like in the following photo.



	put	*get*	*space complexity*	$Transpose$
Standard implementation	$O(1)$	$O(1)$	$O(r)$	$O(rlogr)$

Corollary. *We don't need to do the operation on the matrix itself, it's enough to convert the input.*

modified implementation of Transpose opearation. Instead of converting the matrix, we will hold a flag variable $IsTransposed$ which will keep the result if we executed $Transpose$ or not. notice that for every matrix A. $A^{T^T} = A$. in *Get* operation, we will check if $isTransposed$ turned on. if not we access cell (i, j) and if yes, we access cell (j, i). in the same way we implement *put* operation, in $Transpose$ opearation

we will change the flag $isTransposed$. In the following solution time complexity of $Transpose$ is $O(1)$ and the other operation still the same.

	put	*get*	*space complexity*	$Transpose$
Standard implementation	$O(1)$	$O(1)$	$O(r)$	$O(1)$

Special matrices: magic matrices.

Definition. a matrix nxn is called magic, if:

(1) the matrix is full of numbers from $1 - n^2$. every number appear once.
(2) for every two rows i, j, the sum of elements in row i equal to sum of elements in row j.
(3) for every two columns k, l, the sum of elements in column k equal to sum of elements in column l.

Example. Look at the following matrix.

	0	1	2	3
0	16	2	3	13
1	5	11	10	8
2	9	7	6	12
3	4	14	15	1

Exercise. implement a data structure which intialize a empty matrix (with no such a number) and let the player trybto fill the matrix. i.e, you need to support the following operations.

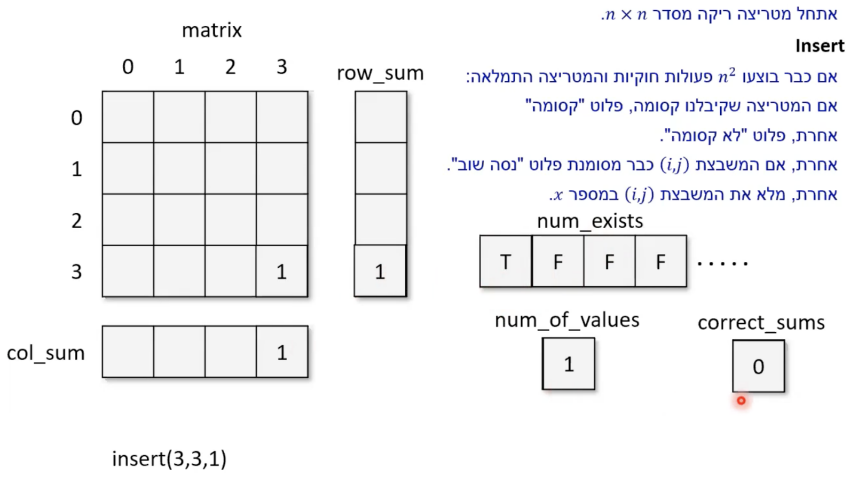
- $init(n)$.
  - initlize a empty matrix with order nxn.
- $insert(i, j, x)$.
  - if we execute n^2 legal operations and the matrix is filled
    - if the matrix filled is magic, we output "magic".
    - otherwise "not magic".
  - otherwise if the index (i, j) is already used we return "try again".
  - otherwise, fill (i, j) by value x.

complexity of the data structure. The required time for the two opearion is $O(1)$. Moreover, space complexity is $O(n^2)$.

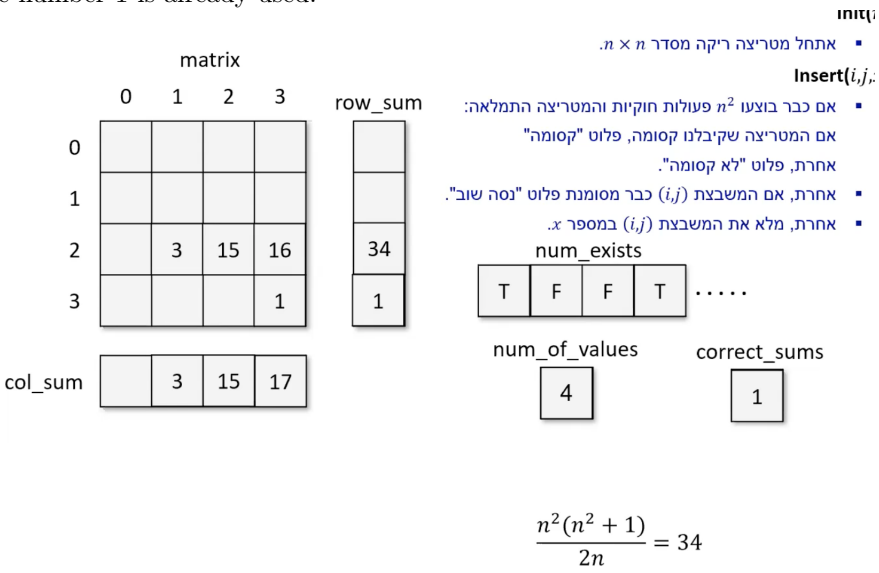
modified implementation. we can allocate a matrix n^2 and two arrays $row\_sum, col\_sum, num\_exists$ which every cell in the follwoing arrays tell as the sum of the row in the matrix respectively E.g row 0 in the matrix sum will be $row\_sum[0]$ and the same for the other rows, now a problem could occur is knowing if the matrix is full by $O(1)$ a naive way is to check all the elements in the matrix and if place is empty we go on otherwise we increment our counter, but the following wat take $O(n^2)$ therefore, we can intialize a counter num_of_values variable in the beginning and each time we insert element we increment the counter by 1 therefore, it takes $O(1)$. num_exists is array allocate in length of n^2 which tell us if the element $1 \leq i \leq n^2$ is used or not if not then we could insert it. now our last problem to check that all the elements in the ow_sum, col_sum arrays are identical, but we can't do that in $O(1)$ since

passing on both arrays elements take $O(n)$. in order to solve that we notice that in each row the sum should be $\frac{1+2+..+n^2}{n} = \frac{n^2(n^2+1)}{2n}$ and we check element in the row sum array each time we insert element to the i, j place (we check sum of row i and sum of column j) if they equal to $\frac{n^2(n^2+1)}{2n}$ if it yes then we increment the new allcoated counter $correct\_sum$ by 1. if it was 1 then we add element to the same row and we have that the sum in the following row is not $\frac{n^2(n^2+1)}{2n}$ then we substract 1 from $correct\_sums$ notice that in the end we should have $correct\_sums = 2n$ in order to satisfy the condition of a magic matrix.

Example. Running example of the data structure proposed above.

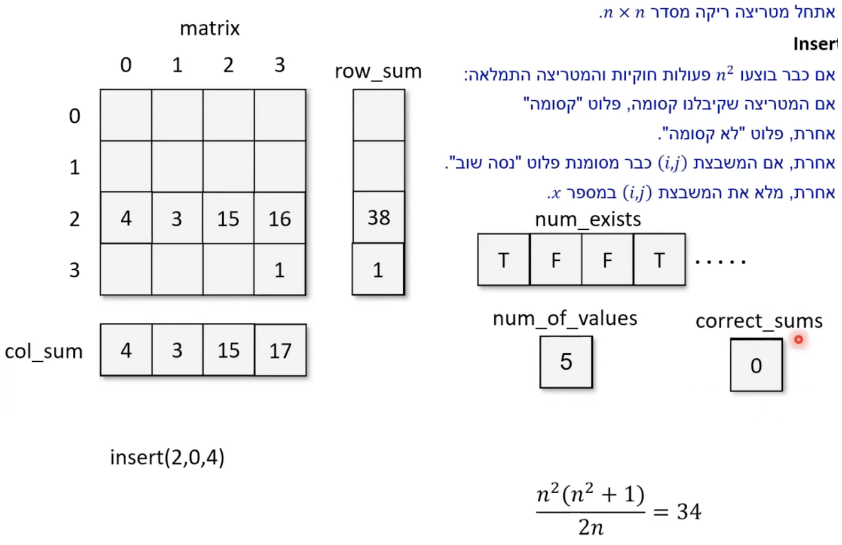


Now notice that if we do again $insert(3, 3, 1)$ we get error that the place $3, 3$ is used since we don't have 0 there Moreover, the array num_exists will give us that the number 1 is already used.



$$\frac{n^2(n^2+1)}{2n} = 34$$

notice that we have on the row number the value of $\frac{n^2(n^2+1)}{2n} = 34$ therefore, we increment $correct\_nums$ by 1.



notice that after $insert(2, 0, 4)$ we see that sum of row 2 is not 34 so we substract 1 from $correct\_nums$ it could never reach $2n$ so it can't be magic matrix at the end of filling the matrix.

Exercise. In the following question we will help a doctor manage his queue dates which were set in his clinic.

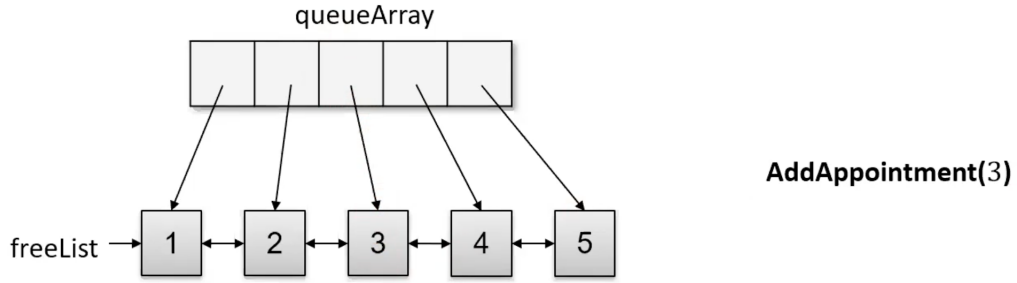
we need data structure which support the following operations -

- $init(m)$ - intializing a appointment celandar with m appointments which are numbered $1, ..., m$ after intializing, all the m appointments are free.
  - time complexity $O(m)$.
- $AddAppointment(i)$ - set a appointment in the date of queue i. if the following data is already used, set appointment in free date. and return the date were determined.
  - time complexity - $O(1)$.
- $CancelAppointment(i)$ **-** cancel the appointment which were determinced in the queue i.
  - complexity time - $O(1)$.
- space complexity - $O(m)$.

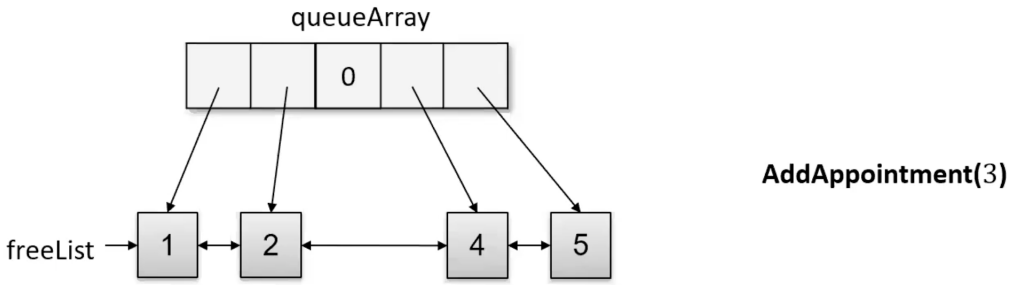
Solution. We can implement the following by using a boolean array of length m and everytime we add appointment we put true in the i place and when we cancel appointment we convert it to false and it's $O(1)$ and we are finished. **But** we have problem occure when we try to do $AddAppointment(3)$ twice, so we can look at $2 \vee 4$ and what if they are also used? and what if the only place is not used is $n = 1,000,000,000$ so we need to pass on all the array elements, so how can we solve it? we can solve that by array of pointers queue $queueArray$ which every element to it point to a node in a directional linked list. now if such a appointment is already added E.g in the place i then in the queuearray we put 0 in the i place and destroy it's pointer to the linked list. notice that linked list has all the free queue. but in case we do the $AddAppointment(i)$ twice as we said above we in the

first time we delete the pointer of the i place in the queuearray and we connect previous node to the nextnode in which the pointer to it is going to be deleted, In the second time we chose a node in the freeList queue which is not used already then we return it. now if we want to do $CancelAppointment(i)$ we can solve it in two cases, first to allocate a node again between $node(i-1), node(i+1)$ and make $queueArray[i]$ point to it but in this case, if we have a lot of elements we are going to deal with a $O(m)$ time complexity in the worst case instead of $O(1)$, so we can solve it by inserting node in the head of the freelist and appointment i and make $queueArray[i]$ point to it.

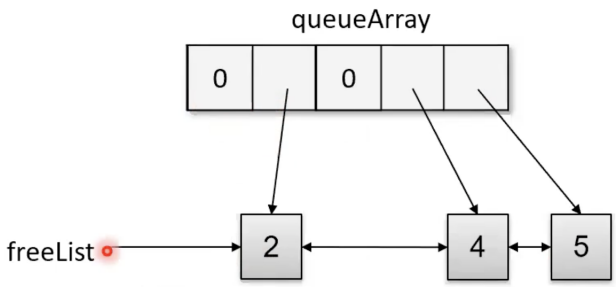
Example. Running example of the data structure above.



as in the picture we can see how our data structure look like, freelist is a directional linked list and queueArray is a array with m places. threfore $O(m)$ space complexity.

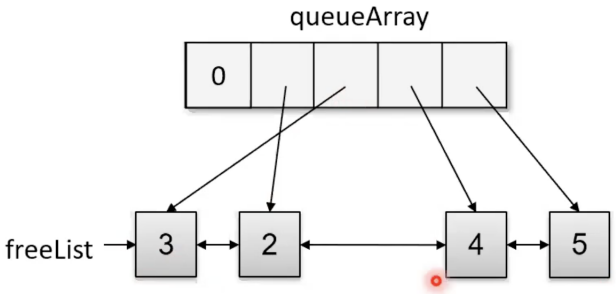


After we add appointment we delete the pointer and put 3 in the queue array at the place 3.



AddAppointment(3)

After we do *addAppointment*(3) again we look for not used place in the queue in this case we can see that 1 is not used, then we use it so we took a random appointment in a free data.



CancelAppointment(3)

CancelAppointment(3) after we cancel it we don't need to allocate node again between node with queue number 2 and 4 so we can allocate new node as head and point queue number 3 to it as in the photo above instead of passing to many places in the linked list which could take $O(m)$ in the worst case so we avoid it by allcating the head node and point to it by the queueArray[3].

Remark. notice that there is no meaning of the nodes order in the following data structure by it's important that the nodes are still connected in such a way with no order E.g $3 \longleftrightarrow 2 \longleftrightarrow 4$....

AVL Trees

Balanced Trees & The Relation To Fibonacci Trees.

- Complete tree is absoulately a balanced tree. However. a lacing in not balanced.
- the problem is to prserve the property of balanced tree while inserting and deleting element each time.
- how are we going to define a balance tree.
- our target is low height.
- as regular: asymptotic definition, logarithmic height.
- we will define what a balanced trees are.
- we will define AVL trees.

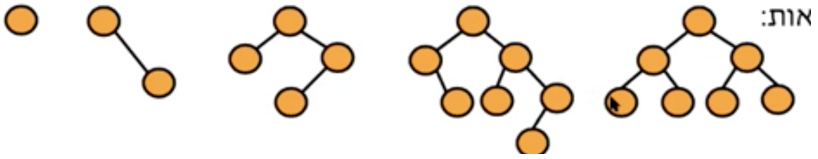
- we will build a a algorithm to insert and delete elements while preserving the balance property.

Definition. we deonte by $h(T)$ the height of tree T. the family of trees is called balanced if exist function $f(n) = O(logn)$, and for every tree T with n vertices in the family satisfied that: $h(T) \leq f(n)$.

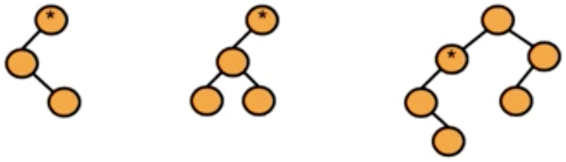
Definition. AVL tree is a searching binary tree which for every vertice v satisfy the property:

$$|h(v \to left) - h(v \to right)| \leq 1$$

Example. Examples of AVL trees:



Example. Example of non AVL trees:



Intiuition.

- The definition seem balancing.
- We will show that for AVL tree in height h, number of vertices n is at least $n > 1.6^h$.
- from here $h = log_{1.6}n = O(logn)$.

Proof balance of AVL trees.

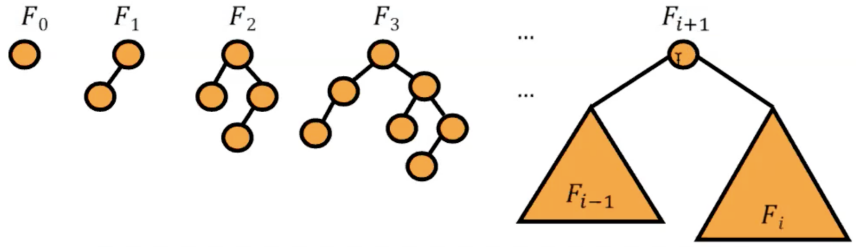
- we are going to define family of trees - Fibonacci trees $F_0, F_1, ..., F_h$.
- they are going to be AVL trees "the most unbalanced" and we will show that they still balanced explicitly:
- we will show that for every AVL tree with height h there is number of vertices bigger or equl to $|F_h|$.
- We will show that the height of AVL tree F_h is h, and the number of vertices in F_h trees satisfy $|F_h| \geq a^h$ for $a > 1$ parameter.

Corollary. *we conclude that:*

- *every AVL tree with n vertices and height h, satisfy that* $n \geq |F_h| \geq a^h$.
- *for every AVL tree with n vertices and height h, satisfied that* $h \leq log_a n$.
- *AVL tree is balanced.*
- *Fibonacci trees are AVL trees the most unbalanced, in which their height grow the most quick as the number of vertices.*

Fibonacci trees.

Definition. we are going to define family of fibonacci trees in recursive way:



Lemma. *$\forall h$, the height of fibonacci tree F_h is h.*

Proof. by induction on h :
Base: it's true for F_0 and F_1.
Induction step: satisfied by the definition of fibonacci tree:

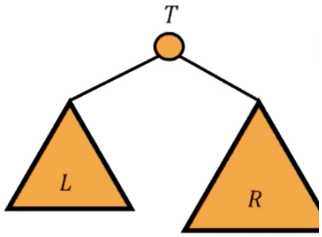
$$height(F_{i+1}) = height(F_i) + 1 = i + 1$$

□

Lemma. $|F_h| = 1 + |F_{h-1}| + |F_{h-2}| \geq |F_{h-1}|$.

Lemma. *let T be AVL true. then $|T| \geq |F_h|$.*

Proof. by induction.
Base: let T be AVLA tree with height h.
let L be the left subtree and R the right subtree as in the following picture.



- The subtrees R, L are AVL trees with small height than T.
- one of them WLOG R the right subtree is with height $h-1$ and the left if $h-2$ or vice-versa.
- by the induction assumption, $|R| \geq |F_{h-1}|$
- by the induction assumption and lema 2 :

$$L \geq min\{|F_{h-1}|, |F_{h-2}|\} = |F_{h-2}|$$

- bu the following T is a tree which has at least $|F_h|$ vertices and indeed satisfy that since:

$$|T| = 1 + |R| + |L| \geq 1 + |F_{h-2}| + |F_{h-1}| = |F_h|$$

□

Corollary. *we concluded that:*

- every AVL tree obtain more vertices the fibonacci with the same height.
- Now we will show that AVL tree obtain exponentional number of vertices

- we will define fibonacci number.
- we will show the they grow exponentialy.
- we will show that the number of vertices in fibonacci tree is fibonacci number.

Fibonacci Numbers.

Definition. fibonacci sequence is defined in the following way:

$$n_0 = 0, n_1 = 1$$

when the other sequence elements is defined by:

$$n_{i+1} = n_i + n_{i-1}$$

Example. first elements in the sequence,

$$n_0 = 0, n_1 = 1, n_2 = 1$$

$$n_3 = 2, n_4 = 3, n_5 = 5$$

$$n_6 = 8, n_7 = 13, n_8 = 21$$

Lemma. $n_i = \frac{\phi^i - \bar{\phi}^i}{\sqrt{5}}$ *when* $\phi = \frac{1+\sqrt{5}}{2}, \bar{\phi} = \frac{1-\sqrt{5}}{2}$ *(ϕ is called the gold relation).*

Remark. Since $|\bar{\phi}| < 1 < |\phi|$ stem that $n_i \approx \frac{\phi}{\sqrt{5}}^i$ for big enough i.

Proof. (lema 4).

Given the following formula

$$n_{i+1} = n_i + n_{i-1}, n_0 = 0, n_1 = 1$$

assumimng that the solution is in form

$$n_i = x^i$$

we plug-in into the formula and get that

$$x^{i+1} = x^i + x^{i-1}$$

deviding both sides by x^{i-1} give us:

$$x^2 = x + 1$$

Therefore the solutions for the following formula is:

$$\phi = \frac{1+\sqrt{5}}{2}, \bar{\phi} = \frac{1-\sqrt{5}}{2}$$

meaning that $n_i = \phi^i, n_i =, \bar{\phi}^i$ are solution to the equations. Therefore, all it's linear combination is solution. i.e $n_i = a \cdot \phi^i + b\cdot, \bar{\phi}^i$. using intial conditions give us the a, b.

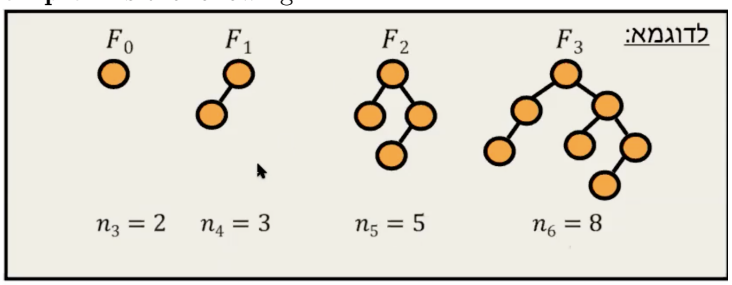
$$n_0 = 0 \Rightarrow a \cdot \phi^0 + b \cdot \bar{\phi}^0 = a + b = 0$$

$$n_1 = 1 \Rightarrow a \cdot (\frac{1+\sqrt{5}}{2}) - a \cdot (\frac{1-\sqrt{5}}{2}) = 1 \Rightarrow a = \frac{1}{\sqrt{5}}$$

hence, the equation solution is $n_i = \frac{\phi^i - \bar{\phi}^i}{\sqrt{5}}$. □

Lemma. *fibonacci trees F_i there is $|F_i| = n_{i+1} - 1$ vertices when n_i is the i - fibonacci number.*

Example. As the following:



Proof. (lema 5),
The following recusrive formula satisfied:

$$|F_0| = 1, |F_1| = 2, |F_i| = |F_i| + |F_{i-1}| - 1$$

Instead of solving this equation directly, we will execute little bit change in order to get the formula which we already solved. let t_i be the number of vertices in F_i+1 i.e $|F_i| = t_i - 1$ we plug-in into the recusrive formula and get that:

$$t_{i+1} - 1 = (t_i - 1) + (t_{i-1} - 1) + 1$$

i.e

$$t_{i+1} = t_i + t_{i-1}$$

$$t_0 = 2, t_1 = 3$$

This equation is the fibonacci number when the intial point is shifted by three indexes, hence, satisfied that:

$$|F_i| + 1 = t_i = n_{i+3}$$

A formal proof is by induction on i:
Base:

$$t_0 = 2 = n_3, t_1 = 3 = n_4$$

Induction step:

$$t_{i+1} = t_i + t_{i-1} = n_{i+3} + n_{i+2} = n_{i+4}$$

□

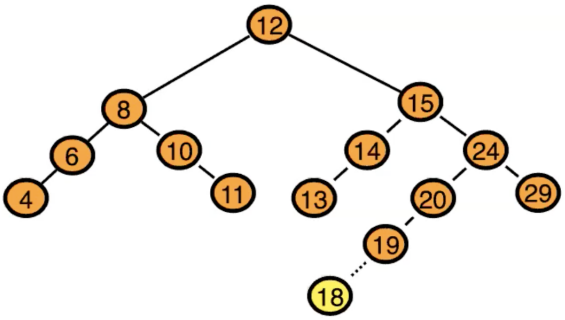
Lemma. *let T be AVL tree with n vertices and height h, then $h = O(logn)$.*

- *By lema 3 we have that $n = |T| \geq |F_h|$.*
- *By lema $4, 5$, we have that:*

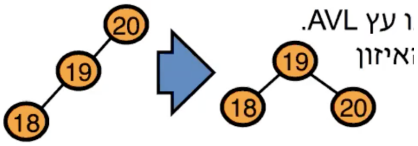
$$n \geq |F_h| = n_{h+3} - 1 = \frac{\phi^{h+3} - \bar{\phi}^{h+3}}{\sqrt{5}} - 1 \geq_{1 > \phi^{h+3}} \frac{\phi^{h+3}}{\sqrt{5}} - 2$$

- *Logarithm of both sides give us:*
$$log_\phi(\sqrt{5}(n+2)) \geq h+3$$
- *isolating h give us:*
$$log_\phi(n+2) + log_\phi\sqrt{5} - 3 \geq h$$
- *In total we have that:*
$$h = O(logn)$$

preserving balance while inserting and deleting. from lema 6 we conclude that the time take of searching in AVL tree is $O(logn)$, we need to insure that after insertion or deleting the tree still AVL.



notice that after we add 18 we got a non AVL tree. but we can make a change in the subtree in which the balance were distorted in that way:

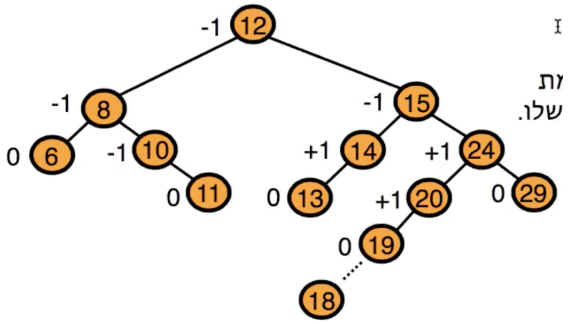


this correction is called rotation. in the deleting elements the same problem could occur E.g deleting 29 from the tree in the photo above.

Definition. for every v in binary tree.

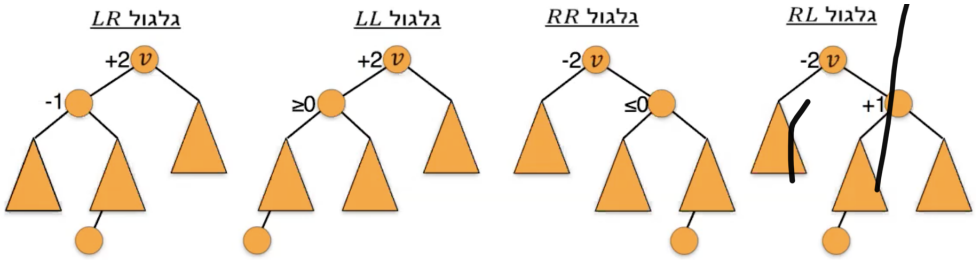
- $h_L(v)$ - is the height of the left subtree of v.
- $h_R(v)$- is the height of the right subtree of v.
- The balance factor is calculates as $BF(v) = h_L(v) - h_R(v)$.

Example. Look at the following tree, each balance factor of vertice is written left to it.



- the only vertices in which the balance factor were distorted within insertion/deleting.
- if for vertice v in the following path the height of the tree in which it's root is v haven't changed then the balance factor on it in the pass didn't changed.
- if the balance factor changed to 2 or -2. then we are required to do rotation in order to keep the tree as AVL. we will fix the distortion from down to up.
- rotation - is a operation executed on the vertice which it's balance were distorted in order to return it to it's legal domain $\{-1, 0, 1\}$.
- balance factor couldn't be bigger than 2 in it's absolute value in every insertion/deletion/rotation it change to 1 at most.

Rotation in AVL trees. the rotation, i.e the way to fix unbalance distortion in vertice, depend on the way in which it was distorted. we always look at the lowest vertice which is not balanced, all it's lineal descendant are balanced. we will categorize the unbalance by 4 different categories which cover all the cases:

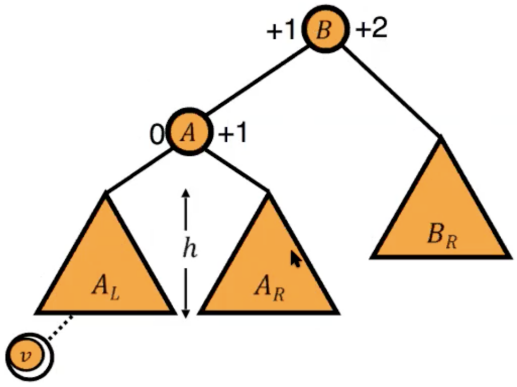


Rotation type	in the right son v_R	in the left son v_L	in the root v
LL		$BF(v_L) \geq 0$	$BF(v) = 2$
LR		$BF(v_L) = -1$	$BF(v) = 2$
RR	$BF(v_R) \leq 0$		$BF(v) = -2$
RL	$BF(v_R) = 1$		$BF(v) = -2$

notice that in the root in which the balance were distorted the balance factor $\in [-2, 2]$ but the left and the right son could be a vertices which there balance distorted but it's not neccessarily E.g in the photo above under LL rotation the following tree has a root with balance factor 2 and it's left son could be with balance factor 0 by adding other vertice under the right triangle vertice or it could be more than 0 by adding other vertices under the son of left triangle vertice so we can always add in such a way that out $BF(v_L) \geq 0$ the same for RR rotation. notice that the root has balance factor of 2. notice that in LR we can only get the result of balance factor $= -1$ since in case balance factor $= 0$ for left son we solve it by the LL rotation so the case in which we have balance factor $= 0$ in left son or right son we solve it by LL, RR rotations respectively.

LL Rotation.

- before inserting: our tree height is $h + 2$. a vertive v were inserted which make tree A_L height $h + 1$.
- LL rotation: we pass A to the root.



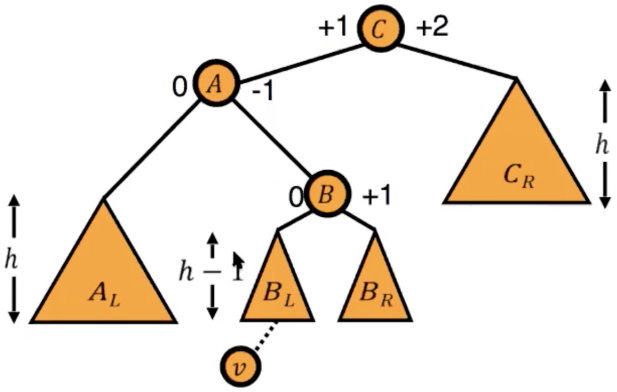
- after inserting v : the tree height after the rotation is $h+2$. like before inserting all the vertices in the tree are balanced. notice that we changed $O(1)$ pointers therefore the time rotation took is $O(1)$.

LR rotation.

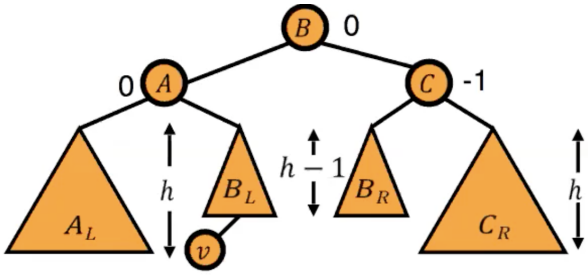
- before inserting v: the tree height is $h+2$. a vertice v were inserted which increase the height of B_L to h.

  Remark. notice that before it was $h-1$.

- LR rotation: passing B to the root.

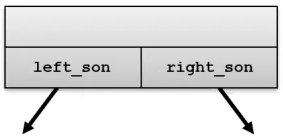


- after inserting v : height of the tree after the roatation is $h+2$ as before inserting. all the vertices in the subtree are balanced. we changed $O(1)$ pointers therefore, the time which rotation took is $O(1)$.



Binary Trees Implementation & Exercises.

implementation.

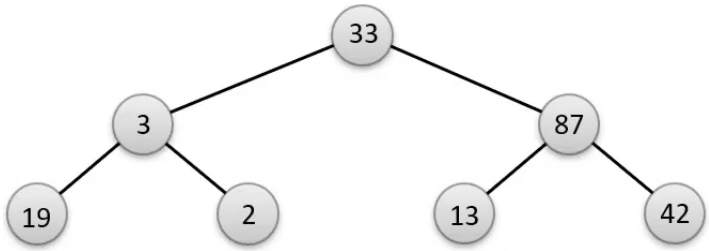



```
struct node {
   int            key;
   void*          info;
   struct node*   left_son;
   struct node*   right_son;
}
```

Remark. notice that:

- every nodes has a unique key which make it different than other nodes.
- if there is no left or right son, then the pointer value is NULL.
- we could also add to every node a pointer to it's parent.

Reminder.



- Preorder tranversal:
  - visiting root.
  - Preorder tranversal in the left subtree.
  - Preorder tranversal in the right subtree.
  - result: 33 3 19 2 87 13 42.
- Inorder tranversal:
  - Inorder tranversal in the right subtree.
  - visiting root.
  - Inorder tranversal in the left subtree.
  - result: 19 3 2 33 13 87 13 42.
- Postorder tranversal:
  - Postorder tranversal in the left subtree.
  - Postorder tranversal in the right subtree.
  - visiting root.
  - result: 19 2 3 13 42 87 33.

Inorder implementation.

- With recursion:

```
inorder(node* p) {
    if (p == NULL) return;

    inorder(p->left);
    do_something(p);
    inorder(p->right);
}
```

- with no recursion (with stack).

```
ptr = root;
while (1) {
    while (ptr != NULL) {
        push(ptr);
        ptr = ptr->left;
    }
    if (is_empty()) break;

    ptr = pop();
    do_something(ptr);
    ptr = ptr->right;
}
```

explaination.

first we point out pointer to the root, then we enter $33, 3, 19$ *to the stack. now notice that for the node which have* 19 *as value first we save it then we delete it from our stack and print it by the saved pointer, then we look at the right son of the node it is null then the interior loop won't be executed and we have* 3 *in the head of the stack, now we do pop from* 3 *then we look at it's right son which is leaf and has value* 2 *we push it to the stack, then we when we look at it's left son which is null then we are out of the interior while loop and our stack has* $33, 2$ *we pop* 2 *and print it then we do* $ptr = ptr \to right$ *i.e we will have null in ptr since* 2 *is a leaf, and the interior while will not be executed. now in the stack we have* 33 *only we which is the root, we pop it and print it by saved pointer ptr and then we go to the right node of the root which is* 87 *we can see that it's null therefore, we push it to the stack and push also* 13 *now* 13 *left son is null therefore, we exit interior while and we will have* $87, 13$ *in the stack we pop* 13 *and print it and we make ptr point to it's right son which is null, again the interior while will not be executed since ptr is null, and the stack is not empty so we let* $ptr = pop()$ *as before which is*

the head of the stack (87) and we pop it and print it and let ptr point to it's right son 42 *now the stack will have only* 42 *and it's not null therefore, interior while will be executed Moreover, it will stop after the first itirarion since it's a leaf and* $ptr- > left = null$ *so we exit it the while and check if it's empty and the anwer is no since the stack has* 42 *so we make ptr point to it and print it then we delete it and our stack will be empty furthermore, we know that* $ptr = ptr \to right$ *will be null then while will not be executed and when we check is_ empty we will break since it has any elements. hence, the final tranversal gave us* $19, 3, 2, 33, 87, 13, 42$ *as what we expected from inorder.*

complexity of both implementation.

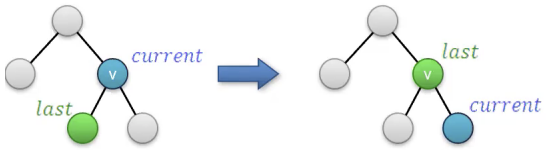
- time: $O(n)$ (with assumption the do_something take $O(1)$.
- space: $O(h)$ (when h is the tree height.

implementing inorder with $O(1)$.. in both implementations of inorder we saw that the space complexity is $O(h)$. how can we execute that with $O(1)$ space complexity with $O(n)$ time complexity? (for the question requirement we assume that every node has pointer to it's parent pointer). now we will hold two pointer:

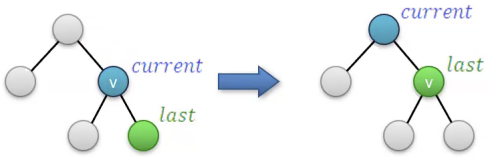
- *current* − which point to the current node in which we exist.
- *last* - it point to the previous node which we was.

now we will do inorder as we did with the stack mathod previously, but instead of storing in stack, we will determine in each step the next step by the place of *last* related to *current*. E.g each time we visit the parent of v, we will check if visited it after the left or right son.

- if we go up from the left son of v :
  (1) we will execute $do\_something(v)$.
  (2) we will continue in the right subree.



- if we go up from a right son of v, it's a sign the we finish tranversal in the subtree of v, hence:
  (1) we will determince currnet to be the parent of v.
  (2) we determine last to be v.



Exercise. in order to back up a binary tree T. it was determined to save the result of tranversal, such in case and the tree were deleted, we could reback it. for each of the following propositions determine whether possilble to reback the tree or not.

- Preorder tranversal.

(1) no, we look at two trees in which preorder give identical result but we can't reback the tree since we can't determine qhether node 2 is a left or right son. as the following:

- Postorder tranversal.
  - no, since the same previous example we will have other result which is 2.1 but we still can't reback the tree. i.e deteming whether 2 is right or left son.
- Inoreder tranversal.
  - no, look at the following trees:

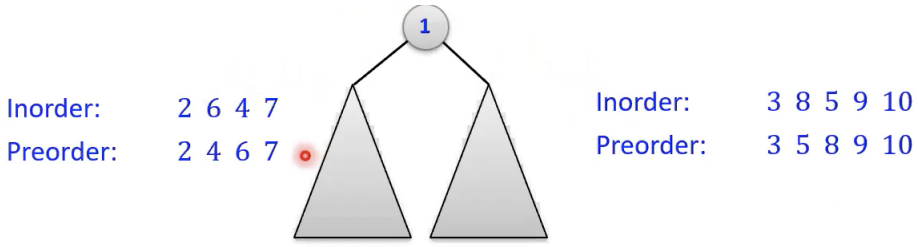
notice that those trees are possible trees of reback but we can't determine which tree were before it was deleted.

- preorder and inorder tranversal.
  - yes, we will see example in which we can reback the tree from those tranversals.

$$inorder - 2, 6, 4, 7, 1, 3, 8, 5, 9, 10$$

$$preorder - 1, 2, 4, 6, 7, 3, 5, 8, 9, 10$$

now we can see that 1 is the tree root since we visit root first in preorder, we look at inorder and see where we have last time 1 appear left to it there is the nodes of the left subtree and right to it the nodes of the right subrtree. now we can look at those subtrees, we know whats it inorder and preorder and we can do it again and again recursively .



now as we see in the photo the left subtree has a root 2 and we can see the this node has a not left subtree but has a right subtree which is $6, 4, 7$ and again and go on so we will have in total 1 is a root and has a node 2 as left son which has a right son 4 and 4 node has a left son 6 and right son 7. identical way to find the right subtree of 1.

- could we reback the tree using postorder and inorder?
  - yes, now we know that the root is the last node of postorder tranversal.
- could we reback the tree using postorder and preorder?

– no, we already saw example which preoder give us $1, 2$ and postorder $2, 1$ of both trees and we couldn't able to determine which one is the original tree, since we can't know whether 2 is right or left son of the root 1 but notice that we can know what the root is in both trees.

Binary search tree. Dictionary data structure support the following operations:

- $Find(x)$ - search if x is in the dictionary.
- $Insert(x)$ - insert x to the dictionary.
- $Delete(x)$ - delete x from the dictionary.

We can implement dictionary data structure using binary searching tree.

Definition. searching tree is a binary tree which satisfy the following proberties:

- every node in the right subtree has a bigger key.
- every node in the left subtree has a smaller key.

complexity time of the three operations Find, Insert, Delete in searching tree is $O(h)$..

- worst case: $O(n)$, when a tree is a laced.
- good case: $O(logn)$ when it's full tree.
- avarage case: $O(logn)$.

Exercise. given a two binary trees, each one has n nodes. descirbe a algorithm which merge those tree to a searching tree with minimal height.

naive solution: walk on all nodes of one trees and insert it to the other tree, in total we have $O(n^2)$ *in the worst case. the tree height in worst case is* $O(n)$..

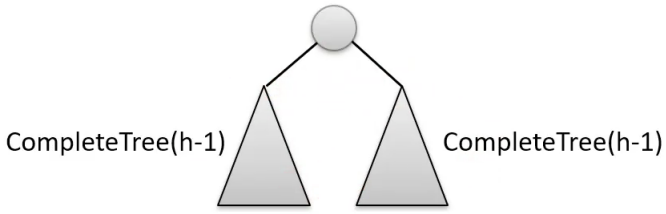
Solution. in order to improve our algorithm. we will exploit the fact that both trees are a searching trees, therefore inoreder tranversal will give us then tree element in sorted way (increasing sorted).

- we walk on both trees by Inorder tranversal and we insert there element in incresing way to two arrays A, B: $O(n)$.
- we union those array to third array C with length of $2n$ by merge algorithm: $O(n)$.
- we build almost full tree with length $2n$: $O(n)$.
- we do Inorder tranversal on the new tree and we insert to it C elements in incresing way $O(n)$.

construction of a complete tree.

Remark. not for every n we can build a complete tree.

we can build a compleyr tree with height h using the following recusrive:



reminder:

(1) number of nodes in a complete tree is $n = 2^{h+1} - 1$,
(2) height of a complete ree as number of nodes is $h = log_2(n+1) - 1$.

- only of $n+1$ is a power of 2, we can do it.
- In the original question, we wanted to build a tree with a even number $2n$ of vertices, which is not possible for complete tree.
- what is the smallest complete tree in which number of nodes is bigger than $2n$? tree with height $h = \lceil log(2n+1) \rceil - 1$, if it could be a complete tree, then it's number of nodes was:

$$N = 2^{h+1} - 1 = 2^{log(2n+1)-1+1} - 1 = 2n + 1 - 1 = 2n$$

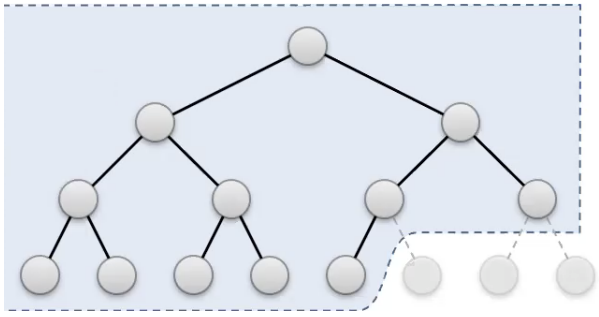
reminder.

Definition. lmost complete tree is a tree in which (maybe) we omitted some leafs from it's down level, starting from the most reight leaf.

- how many leafs we have more than what we require in the smallest full tree which has number of nodes bigger than $2n$: $(2^{h+1} - 1) - 2n$.
- E.g if want a almost complete tree with 12 nodes. how many unwanted leafs we have in the smallest full tree which has number of nodes bigger than 12?

$$(2^{([log(2n+1)]-1+1} - 1) - 2n = 2^4 - 1 - 12 = 15 - 12 = 3$$

$\overbrace{\quad}^{2^{h+1}}$



Constructing of almost complete tree with N nodes.

- We build the smallest tree which has number of nodes bigger than N.
- We will execute opisite inorder tranversal, in which we visit first the right subtree then the left subtree.
- The function check if we reach leaf in tranversal, if yes - then delete it.

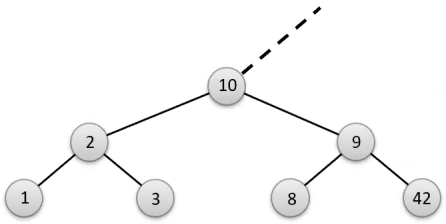
Exercise. Given a binary search tree, and given that exist exactly one node which doesn't satisfy the seaching property. in other words, exist v with key x, S.T in the left subtree of v exists valuse which are biggen than x, or in the right subtree exists a values which are less than x.

propose a algorithm in which can convert the tree to a correct searching tree in $O(n)$ in the worst case.

Solution. The following algorithm:

- We will execute a Inorder tranversal in order to find the invalid node - $O(n)$.

- We will use the algorithm in previous question in order to creat a valid binary search tree from the two subtrees of the invalid node - $O(n)$.
- We insert the invalid node to the tree we created - $O(n)$.
- We change the subtree in which has root v in the created tree.

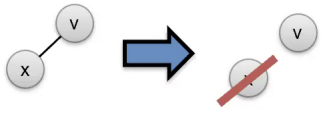


notice that 10 has 9 in the right subtree therefore 10 is invalid node, now using Inorder will guve us the binary search tree in sorted way, so we can idenetify that 10 is invalid node. now we use previous algorithm which merge those subtrees in $O(n)$ then we insert 10 to it in legal way then we make the pointer to 10 point to the new tree root.

Deleting node form a binary seach tree. We need to delete the node which has key x. we will denote it's parent by v.

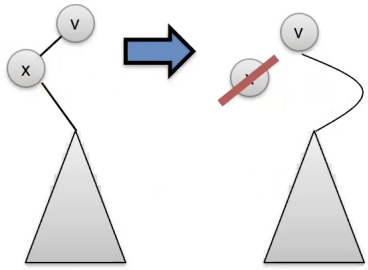
we will split into three cases:

(1) x is a laef: we will delete x from the tree and finish.



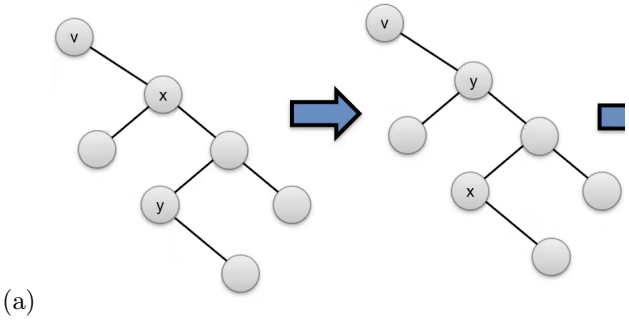
(a)

(2) x has a single child: we will conect the son of x to v in the folowing way:

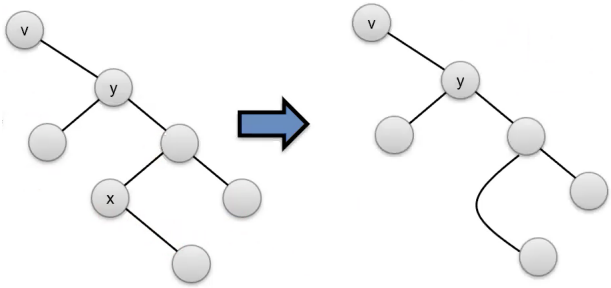


(a)

(3) x has a two childs: we will change x with it sequential element to x (recall it y).
   - the sequential element to x : the smallest element which is bigger than x. i.e the next element in inorder.
   - if x has two childs, y is the minimal element in the rightsubtree of x.
   - now x has a single son at most (if y has a left child, then it wasn't the sequential element) we will delete x as we describeed before.



(a)

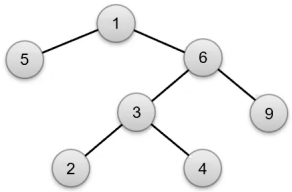


(b)

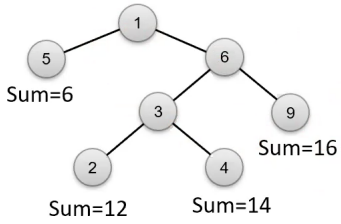
Finding the most weighted path in a binary tree (not neccissarily searching tree).

Exercise. Given a tree with n nodes, in which every node has key and two pointers for it's child. propose a algorithm to calcualte the most weighted path from the root to a leaf (the path in which the sum of it's nodes is maximal) which run in $O(n)$ complexity. the algorithm should print the follwing path.

Example. look at:



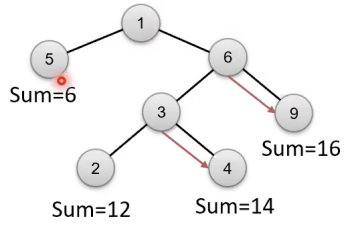
Notice the here $sum = 16$ is the most weighted path. since:



Solution. we will execute Postorder tranversal in which we will return from every node return to it's parent the most weighted path in it's subtree. in order to calculate the most weighted path of node v, we will choose the most maximal weighted path from the two values which the children return adding to the value v(or the only path in case v has a single child, or the weight of v only if v is a leaf).

how are we going to return the path?

we will add a other field which is a pointer for every node we visit.



intiuition to previous question.

for finding the most weighted path in the graph, first we do postorder which check the left subtree first then root then right subtree, now in the graph above we first visit the left subtree we see 5 and it has no sons therefore we return it's value and wait tell we finish with the right subtree, now in the right subtree we visit 6 first and go deep into it left son which is 3 now 3 has two sons we go to 2 we see that it's a leaf so we return it and wait tell we check 3 right son which is 4 and we return it too, now to 3 we add the maximal node value which is 4 and node 3 will have $sum = 7$ the same thing we do with it parent which us 6 we check the most weighhed path whether $9 \vee 7$ we see that 9 is the maximal and we add it to 6 so we have $9 + 6 = 15$ and go to the root which will determine we notice that the root has $15, 5$ so the maximal of it is 15 therefore the most weighted path is 15 .now in order to save the path we do pointer to node on each time we choose the maximal son. E.g in this case 6 will point to 9 and 1 will point to 6 and the path we get by walk on the red arrows path which is $1, 6, 9$.

AVL trees. AVL tree also $B+$ which we will see later are examples of a balanced searching trees.

Denote:

- $h_L(v)$ - the left height tree of v.
- $h_R(v)$ - the right height tree of v.
- $BF(v)$- the difference between subtrees height of v.

$$BF(v) = h_L(v) - h_R(v)$$

Definition. AVL trees is a binary searching trees, in which every node satisfy $|BF(v)| \leq 1$.

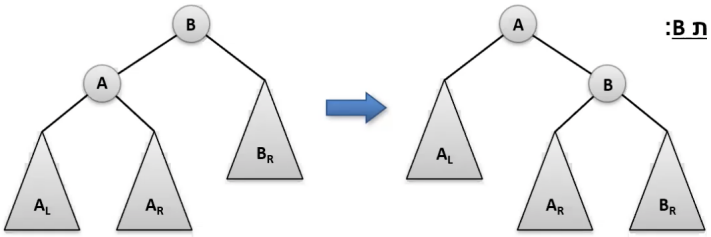
The tree height of AVL (h) satisfy $h = O(logn)$.therefore, the complexity of insert/find/delete is $O(logn)$.

Problem. The insertion operatiom and deleting would exceed the balance factor BF in tree.

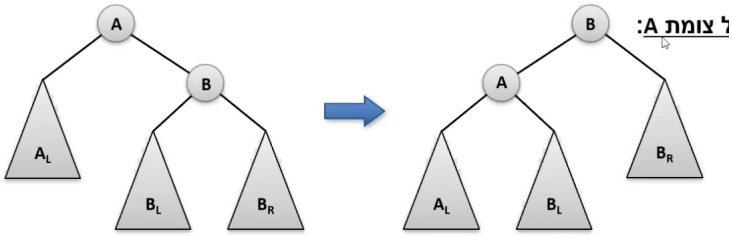
Solution. After inserting/deleting operations we will fix the tree, which will reback it to a undamaged tree. those fixes rely on operation called rotations.

Basic rotations. All the fixing operations of AVL relies on two rotations.

- Right rotation of node B.



- Left rotation of node A.



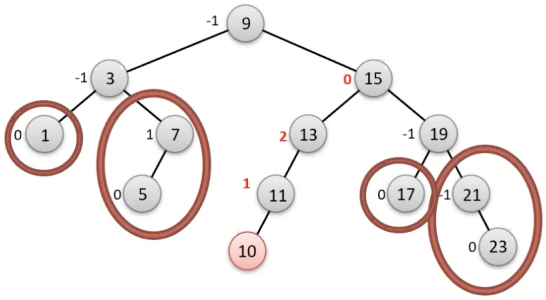
Remark. Notice that both of the operation are symmetric. i.e if we execute a right rotation to B then left rotation to A vice versa, we wil get the same tree.

Remark. rotation preserve the property of binary searching. i.e operating a rotation on tree keep the same binary tree.

Remark. every rotation could be executed in $O(1)$.

Fixing AVL tree.

(1) After every operation of deleting\inserting the factor balance is in $[-2, 2]$. why?
(2) if the balance factor changed to -2 or 2, we are required to execute a compatible rotation in order to switch the tree to undamaged AVL.
(3) The only nodes in which the balance factor were interrupted are the nodes on the inserting\deleting path. since $BF(v)$ is the balnce factor of v. for nodes which are not on the searching path (the path from the root to the new leaf inserted), the subtree under them didn't change.



notice that in the following graph after adding 10 the node which has 13 as value will be damaged since $BF(v \to 13) = 2$.

Is the height og the marked subtrees changed?

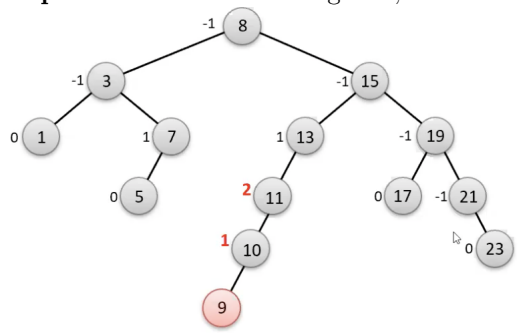
No, hence the way of calcuating BF of there root didn't change also.

How many nodes which there balance factor could be interrupted?

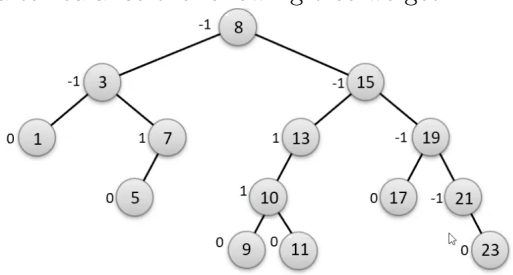
h Since the path is at max has the tree heigh which is $O(logn)$.

(4) if for a node in the following path, if the height of the subtree under it identical to the origin height (i.e before executing deleting/inserting operations) then the balance factor on the nodes above won't change.

Example. look at the following tree,



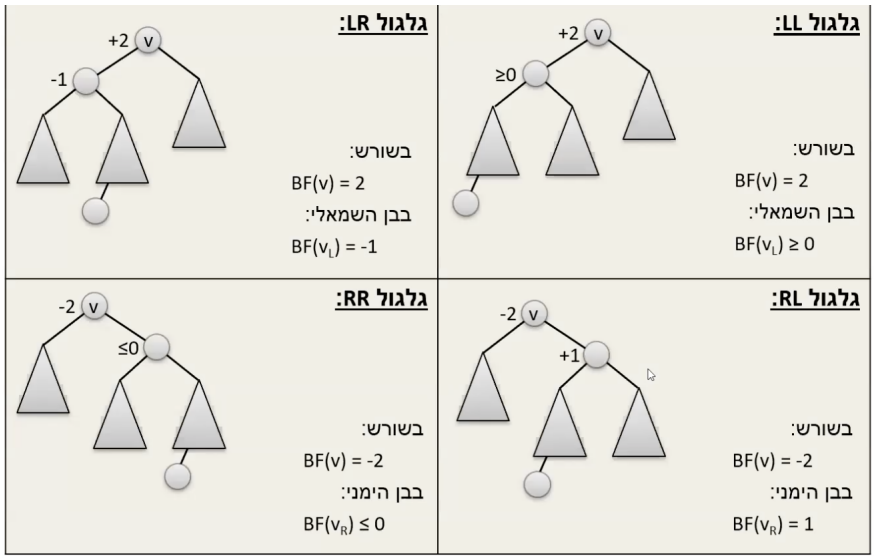
after balance the following tree we get:



Notice that this tree all the node above $v \to 13$ would still with the same balance factor.

Algorithm for fixing a balance factors. For every path searching from the node were inserted/deleted to the tree root.

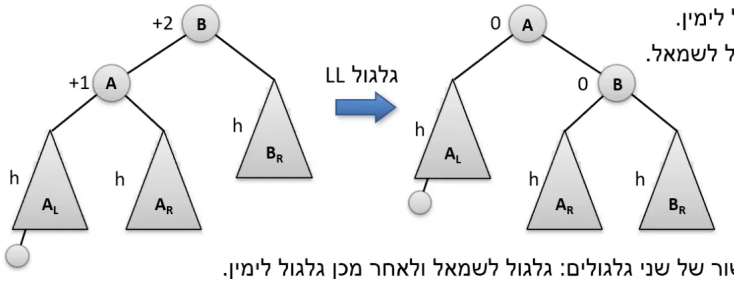
- update it's balance factor.
- if $|BF(v)| = 2$ then execute the compitable rotation.
- if the height of the subtree which it's root v didn't change then finish.
- if the height of the subtree changed and $BF(v)$ is undamaged then go above.



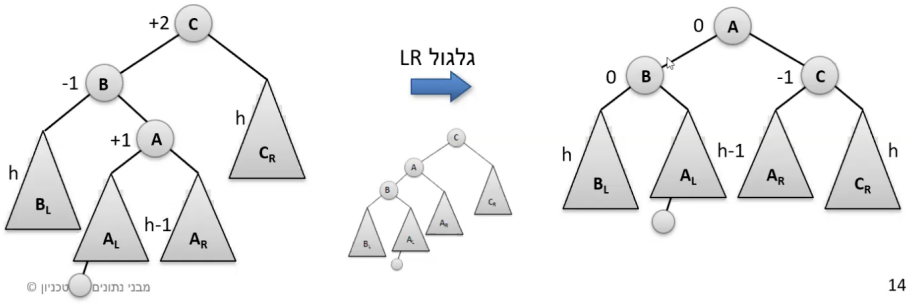
choosing the right rotation.

Rotations after insertion.

- LL rotation is a rotation to right.
- RR rotation is a rotation to left.



- LR rotation is aa sequence of two rotations: rotation to left then rotation to right.



Check that indeed two rotation were executed and we execute LR rotation since B has BF of -1 so that how we knew.

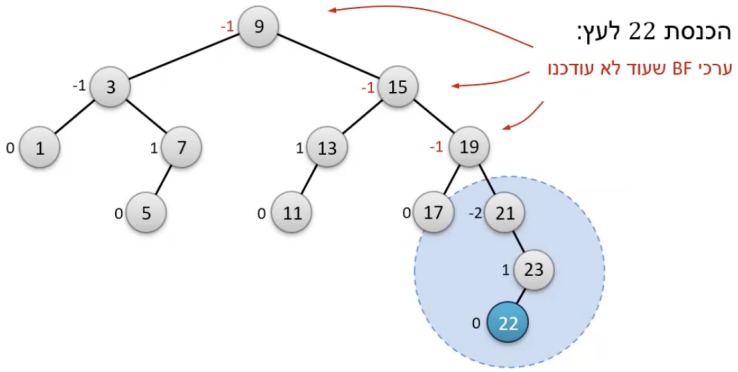
Rotation after insertion. At the four rotation descirbed while inserting, the height of the damaged tree after the rotation is identical to the original tree height. hence, after executing the rotation the tree will be proped.

- The fix of tree after insertion take $O(logn)$.

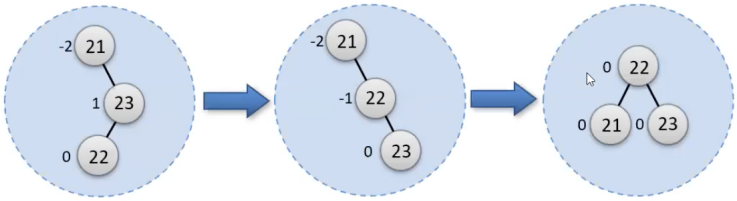
Rotation after deleting. The four rotations are defined in the same way. as opposed o insertion, in most of the cases the tree height of the damaged tree will be changed. hence, after rotation there is a need to go ip in the searching path and check if other nodes were damaged.

- The fix of tree after deleting take $O(logn)$.

Example. Inserting 22 in the following tree.

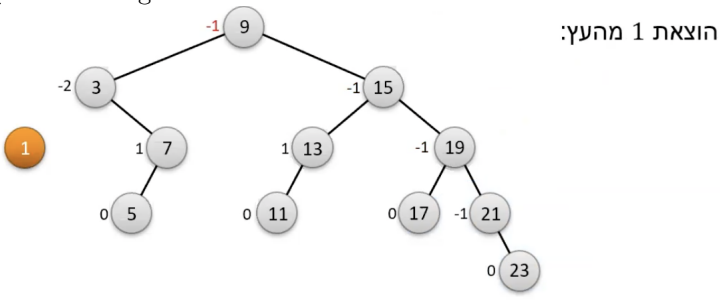


So we need to execute RL rotation:

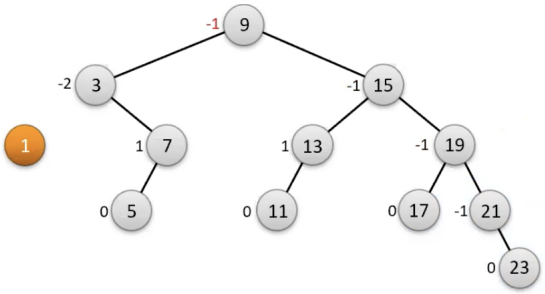


notice that the new subtree height is 1, as height of the old subtree. hence, there is no need to continue in the fixing operation.

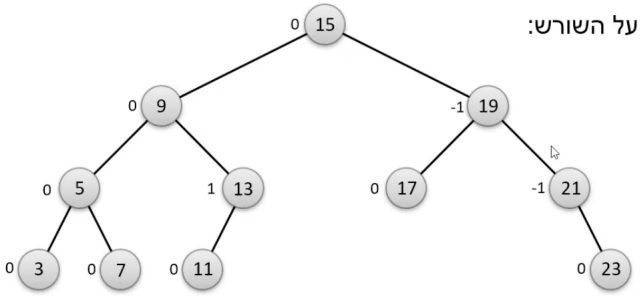
Example. Deleting 1 from the tree above afterward we will have his tree:



Now we see that we need to execute RL on node with value 3.
Update balance factor is written left to each node in the following photo:



Althou we did rotation the tree still no balanced, so we need to execute RR rotation on node which has value 9 i.e the root in order to make it AVL tree. after we do this we get the following balanced tree.

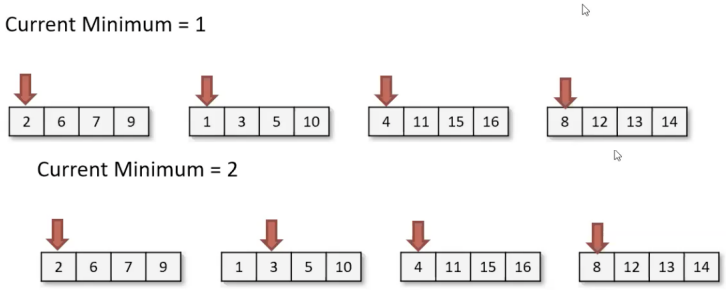


Remark. Notice that we saw the algorithm of inserting and deleting in binary search tree and how we execute it, now after executing them the tree could be unblanaced then we need to do rotation.

Exercise. Given a k searching trees $T_1, .., T_k$ which contain n different numbers. we want to pring the n number sorted in increasing way with time $O(nlogk)$ in the worst case, we can asssume that the trees are not empty.

Naive solution 1. we will print all the elements to a array and sort them in total: $O(nlogn)$..

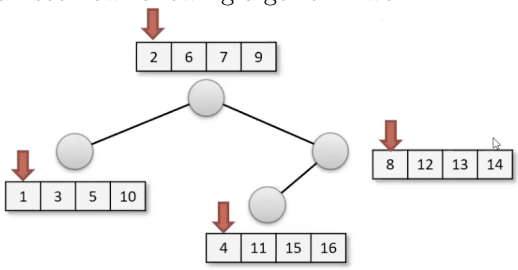
Naive solution 2. executing inorder on every tree to k arrays, then merge them to one array $O(nk)$. *since in every time in merge we comare between k elements and the smallest each time we insert it to the new sorted array.* Noitce that - in every step while merging we compare k elements, although only one changed. we can preserve the order between k elements using AVL tree.



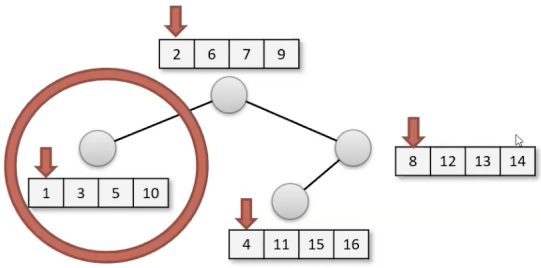
Merging using AVL:.

- Creating a AVL tree sorted by value on which k pointer $p_1, ..., p_k$- $O(klogk)$.
- Finding and deleting the smallest node in the AVL tree - $O(logk)$.
- We will insert the element which point it the output array $O(1)$.
- We will iterate the pointer in the array of node which we were deleted and insert the node again to the tree - $O(logk)$.

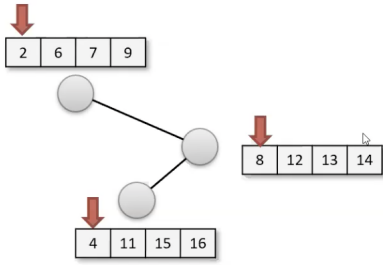
We can see how following algorithm work:



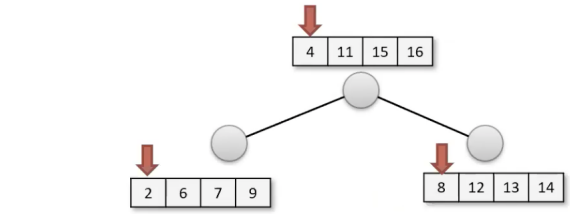
Notice that we have node with the minimal value which is 1 now we delete it and insert it as a node the value 3 to the AVL tree.

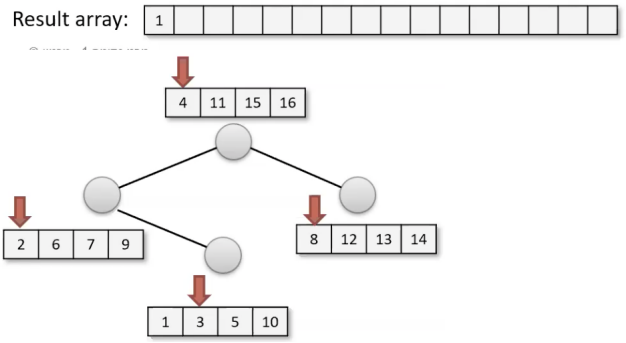


now we delete it and get the following unbalanced tree:



We can see that we have a root with balance factor -2 and rightson with balance factor 1 therefore, we do RL rotation:





Now notice that creating the AVL tree has complexity of $O(klogk)$, but we create it one time, then we finding smallest element\delete\insert element on the smallest node each time and each time finding smallest element\delete\insert take $3 \cdot O(logk)$ since we have n elements we will get $O(nlogk)$ notice that $n > k$ therefore:

$$klogk + \underbrace{3 \cdot O(logk) + ... + 3 \cdot O(logk)}_{n \text{ times}} = O(nlogk)$$

Merging using AVL tree (formal).

- Executing inorder for the k trees each one with n elements in total it will take: $O(\sum_{i=1}^{k} |T_i|) = O(n)$.
- After executing inorder on the k trees we will get k sorted arrays $A_1, .., A_k$.
- Remember that during the merge, for every array there is oiunter p_i which point to the current element in the array.
- We will insert to AVL the k pointer $P_1, .., P_k$, when they are sorted by the heighest value which they point at - $O(klogk)$.
- Now we will fill a array with R the elements in incresing way:
  - in every step we will delete the minimal element P_i from the AVL tree - $O(logk)$.
  - We will insert the element in which P_i point to it to the array.
  - We will iterate the pointer P_i to point on the next element in the array A_i, and we will insert it to the AVL when now it point to a new value in the array - $O(logk)$.
- For every elemeny we add to R, we execute $O(logk)$ operations, hence the running time is $O(nlogk)$.

Exercise. We will say that two binary trees (now neccessarily searching trees) are "equivalence under rotations" if there a sequence of AVL rotations on one of them s.t in the end we get the other tree (in term of topology and keys). Notice that we can operate the rotations on different nodes in the tree and the tree it self are not neccessarily AVL. what is the number of equivalence classes which this relation define?

Solution. since AVL rotations preserve order for Inordeer tranversal. we can find the number of equivalence classes for a specific keys order, and multiply iy by the possible keys order options which is $n!$. if we see that exist topology in which we can get from every topology by rotations, then all the topologies are equivalence under rotations. in this case we ger one class for every possible key order, hence there is $n!$ classes. now notice that key order is matter and different tree with different key order can't reach other under rotations operations i.e T_1, T_2 binary trees, $T_1 \underset{ROTATIONS}{\not\rightarrow} T_2$.

E.g in the following example:

we can see that any AVL rotation can't get us from tree 1 to 2.

But notice that we can convert every binary tree to a unique form of a lacing and each order give as unique lacing form therefore we have $n!$ unique lacing for a each order. now we will show the transform from each binary to a lacing using induction.

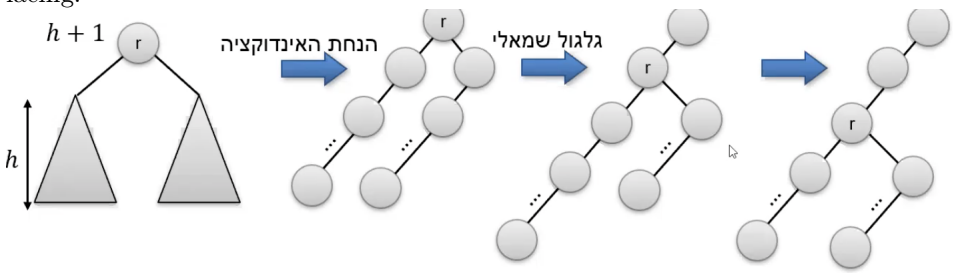
Proof. we will show how to convert each binary tree to a left lacing using a AVL rotations using induction.

Base: for a tree with single vertice, we will get left lacing with no need of rotations.

Assumption: given a binary tree in height h, we assume that exist sequence of a rotations which will convert it to left lacing.

Step: we will look at binary tree in height $h+1$, with root r. and we will split into cases:

- for a vertice there is no right child - we will execute the induction step on the left child and we will get left lacing.
- for a vertice there is right child - we will execute the induction step on the right childs (and left if exist). then we execute the following in order to get lacing.



-

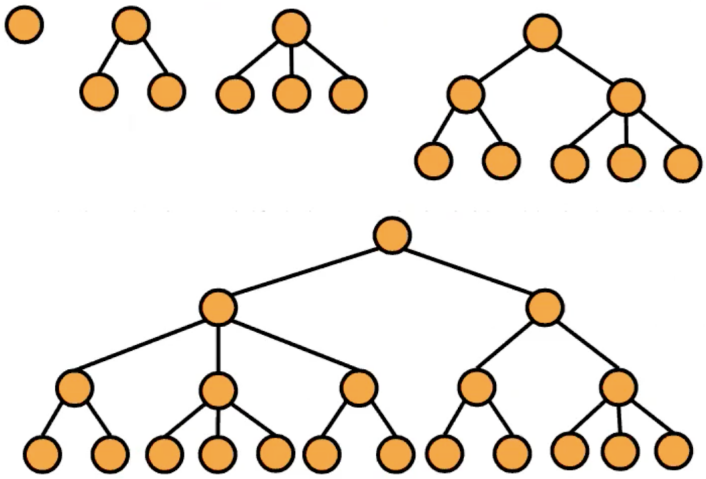
- notice that in the beginning we execute the step induction on left and right subtree of r then we get $r$as a root and left laced subtree to r and right laced subtree of r as we see.
- now notice that after that our induction step doesn't help us getting a one laced tree.
- not if we do left rotation on the tree we execute the step induction on and each time we take one node from right laced subtree of r and put it above r which is left rotation, we keep doing that.
- every step we decrease the height of the right child or $r$by 1.
- we keep doing a left rotation on r till we get a left lace.

□

2-3 Trees.

Definition. $2-3$ tree is a tree in which all the leafs exists at same level Moreover, every interior node has $2-3$ sons.

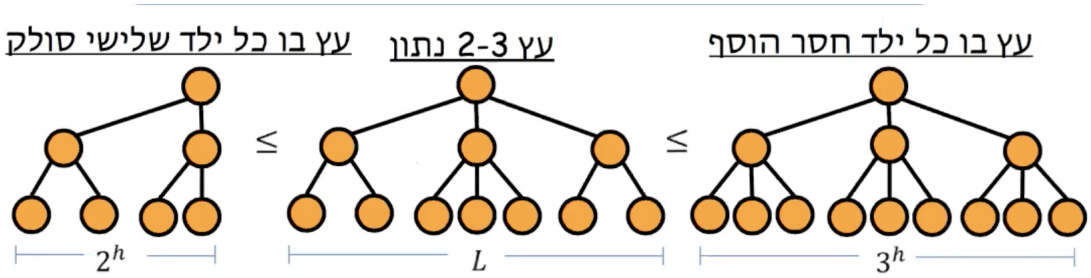
Example. Look at the follwing graph:



Lemma. *The number of leafs in a* $2-3$ *tree satsift that* $2^h \leq L \leq 3^h$ *when* h *is the tree height.*

Proof. for the lower bounder we look at the full binary tree which is produces by omitting all it's thirs child from the $2-3$ tree. and for the upper bouner we look at the tree which is produced by adding thied child to the $2-3$ tree in every place a child is a deficient. □

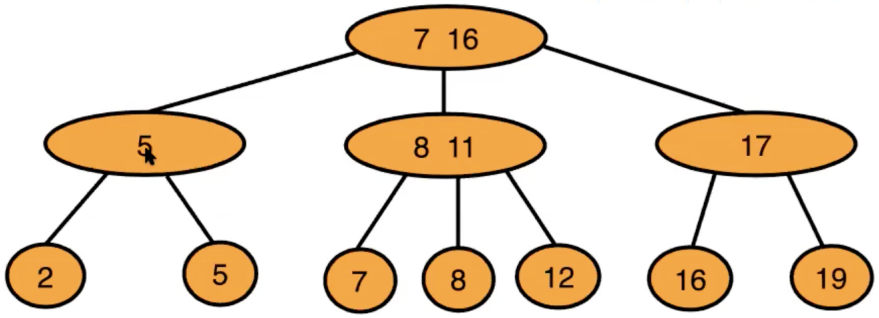
Illustration photo for the proof.



Corollary. *the height of* $2-3$ *satisfy* $log_3 L \leq h \leq log_2 L$ *i.e:*

$$h = \Theta(logL)$$

Preserving data and searching in $2-3$ **trees.** every leaf has a keey and information (we write the key in the leafs) . Every interior node has d sons when $2 \leq d \leq 3$, and in the node $d-1$ indexes which help us to search.



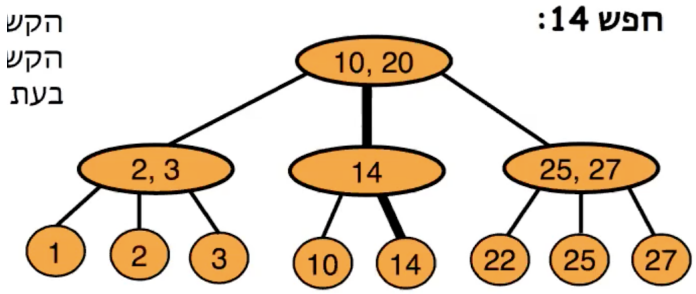
Remark. Notice that in the **interior node with two childs** written a single index k_1 which is strictly bigger than the maximal key in the subtree which is roots is the first and smaller equal to the minimal in the subtree which is root is the second son. **interior node with three childs interior node with two childs** there is two indexes $k_1 < k_2$ the first index k_1 is bigger than the all the keys in the first subtree and less equal than then keys in the second subtree and from k_2. Moreover k_2 is bigger then all the keys in the second trees and less equal from the keys in the third subtree. E.g we can look at the node in the tree above which has $k_1 = 8 < 11 = k_2$ notice that $8 > 7$ which is th first subtree and $8 \leq 8$ i.e less equal from the second subree and $11 > 8 \wedge 11 < 12$ i.e less the third subtree and bigger than the keys in the second subtree.

Searching for a key in $2-3$ *tree.*

(1) let v be the tree root.
(2) if v is a leaf, then search if k exists in the node v. return answer accordingly.
(3) if $k < k_1$ (i.e k is less than the first key of v):
    (a) Continue searching in the first child of v.
(4) otherwisem if v has two childs S.T $k < k_2$ (k is less than the second key of v):
    (a) continue searcihng in the second child of v.

(5) otherwise, continue searching in the third child of v.

Example. E,g searching for 14 in the following tree.

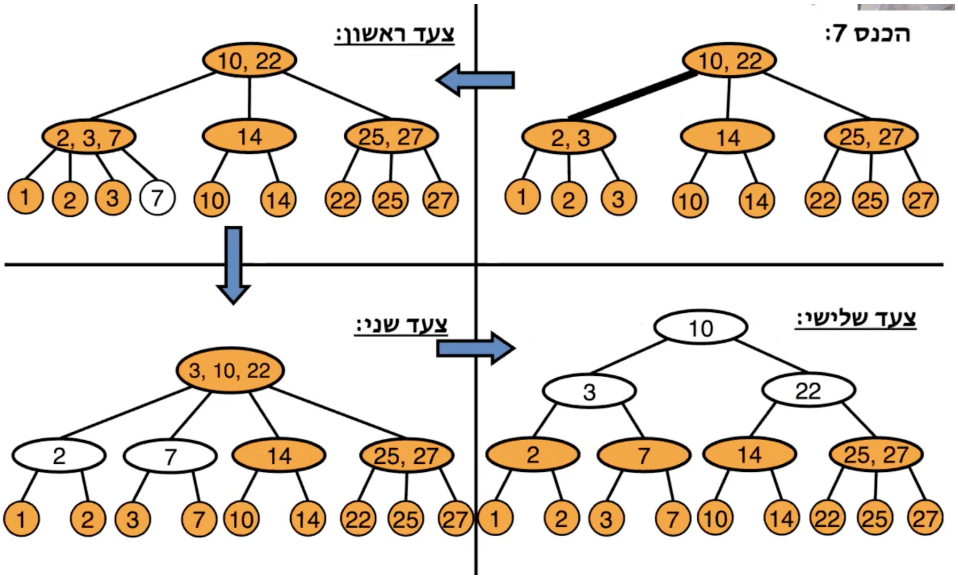


we notice that 14 is bigger then 10 and less then 20 then we search in the second child subtree, in this case the second child has 14 then we complete searching although we have 14 since we want to find 14 in the leafs and 14 in the root son here only help us for searching, it could be 13 oe 12 or other possible values.

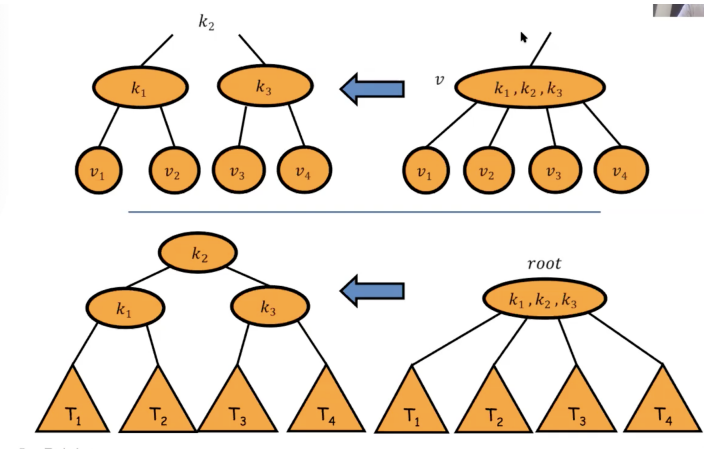
inserting to $2-3$ tree. In inserting to a $2-3$ tree we execute the following operations:

(1) Searching for the place of new key k.
(2) Inserting a leaf to the new place and determince the value of k.
(3) Fixing the tree S.T every node has $2-3$ childs. the fix is made in the new path of k from the leaf which is added to the root. in every level we execute $O(1)$ operations.

Example. the following photo illustrate insertion operation.



explain for splitiing node operation.

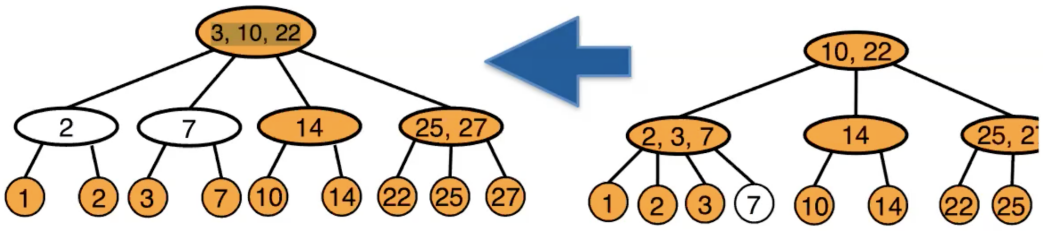


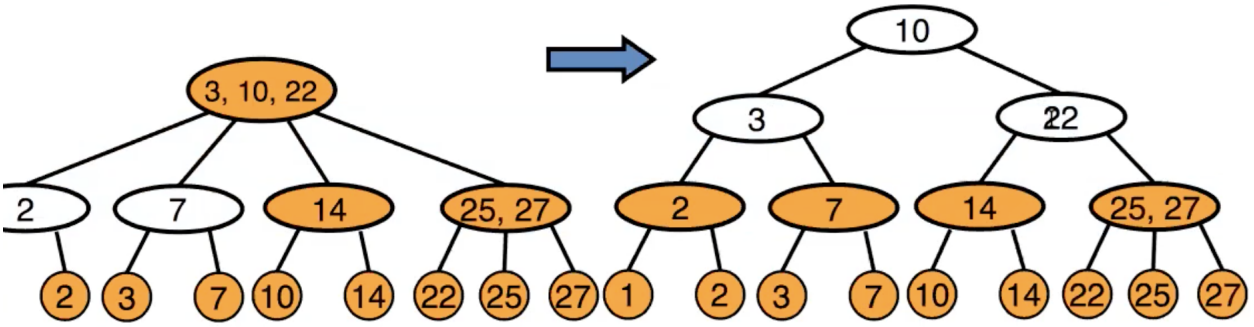
Algorithm for inserting k key. (Part A).

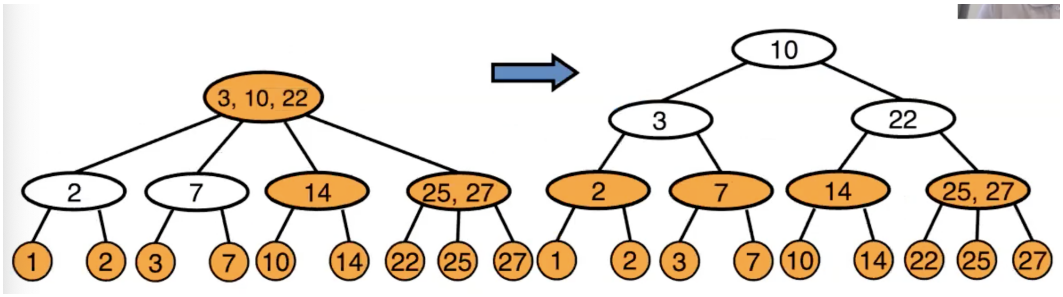
(1) search for k in the tree T. if k exists then finish.
(2) if k is not in T, let f be the last node, which is not a leaf, in the searching path. (if k was in T then f was the parent of the leaf in which k exists).
(3) produce a mew leaf which has k key, and add it as a child to f while preserving the order os f childs.
(4) add to f other index which addapt the $2-3$ tree rules (f may have now 3 indexes).

part(b).

(1) if $v = f$.
(2) if v has three childs, then finish.
(3) if v has 4 childs then split v to a two nodes v_1, v_2.
(4) if v is a root, then allcoate new root f which there childs are v_1, v_2 and finish.
(5) otherwise, let f be the parent of v . connect v_1 and v_2 as childs for the parent f of node $v$while preserving the indexes order, back to step 5.







First we want to insert 7 now we search for it's place as we did before, we add 7 to as third index to the node which has $2, 3$ values and we allocate new node which has 7, now we get problem because we have 4 childs so we solve it by splitting the node we take the middle key which 3 in this case and pass it to the root then the root will have $3, 10, 22$ with 4 childs which are $2, 7, 14, (23, 27)$ when 7 has $3, 7$ nodes as childs and 2 has $1, 2$ as childs. again we have problem becasue the root has 4 childs so we fix it again by splitting we allcoate new node which have the middle key of the root which has 10 and it childs are $3, 22$ when 3 has $2, 7$ as childs and 22 has $14, (25, 27)$ as childs then we are finished.

Remark. notice that we pass index as we did when we pass 3 to the root and get $3, 10, 22$ keys in the root if in case our problem is in the root then we don't pass indexws beacuse we have no parent node for the root. therefore, we allcoate 3 news node and we will have node which is the root stroing our middle key which was in the root and two childs for it.

proberties of inserting progress.

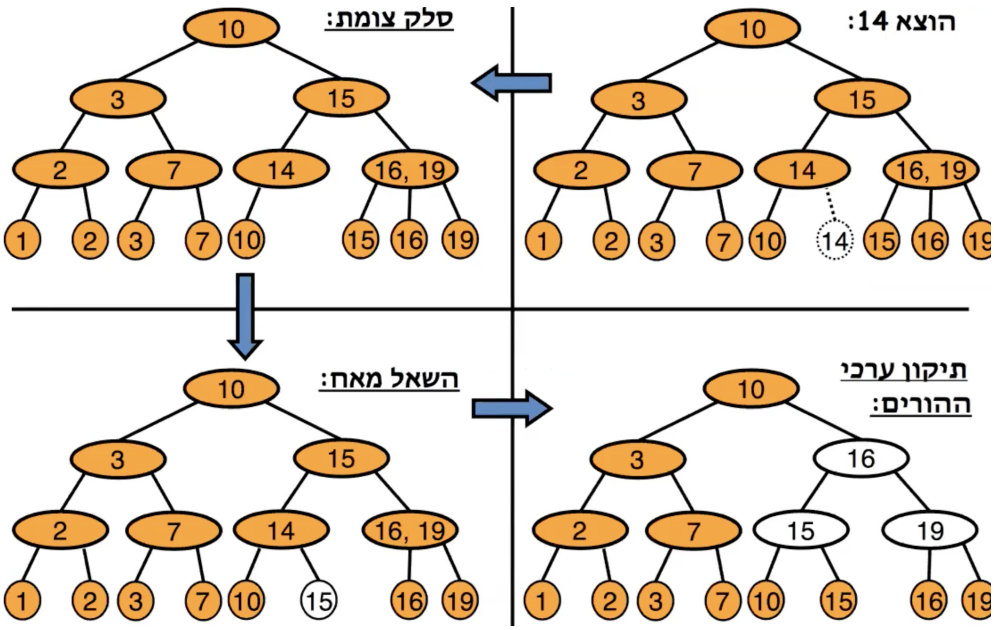
(1) splitting executed only in the path which start from the root to the inserted leaf.
(2) the complexity of each split is $O(1)$.
(3) if 4 nodes were allocated as a childs of a node then we slplit it to two nodes which every one has two childs.
(4) in time of splitting, the new nodes allocated are in the same depth of the node which were splitted.
(5) in splitting the root we allcoate new node which add 1 to the depth of all nodes.

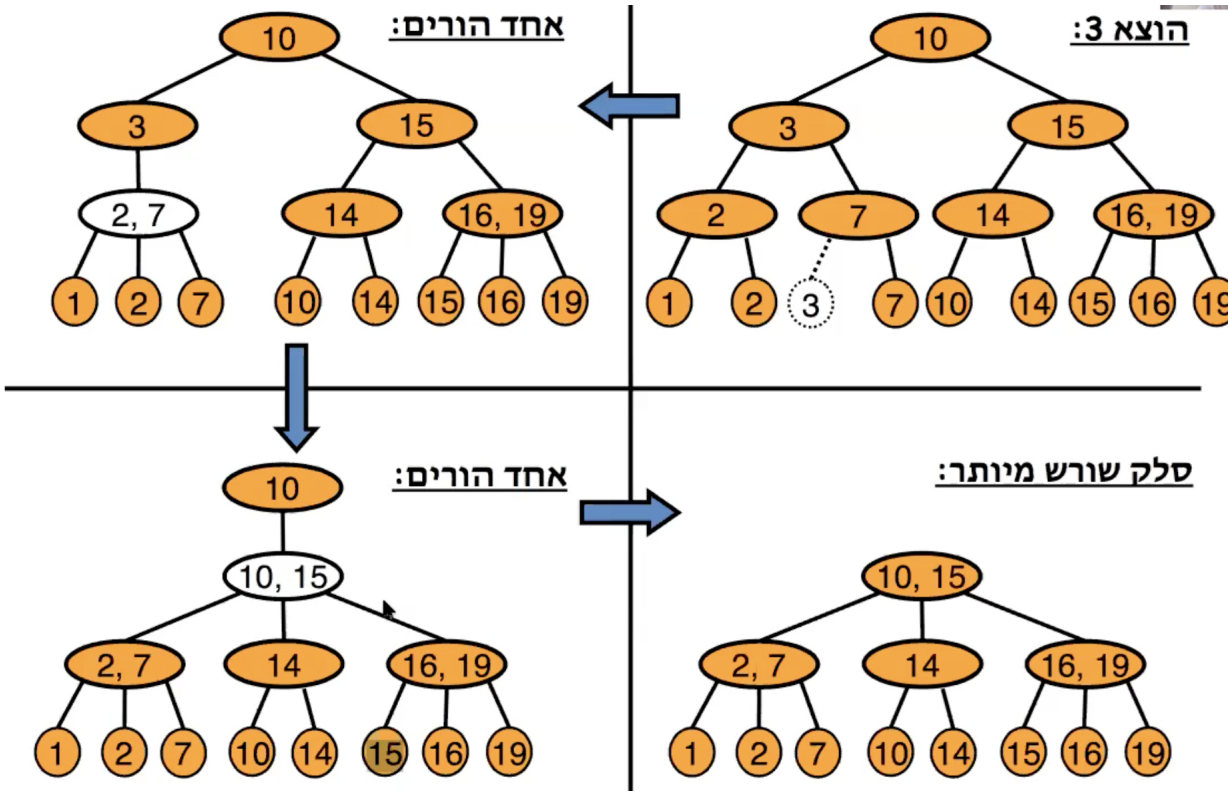
Corollary. *we conclude that:*

- after inserting the tree is proper: all the leafs are in the same depth and every interior node has $2-3$ childs.
- the complexity of inserting is $O(h) = O(logn)$.

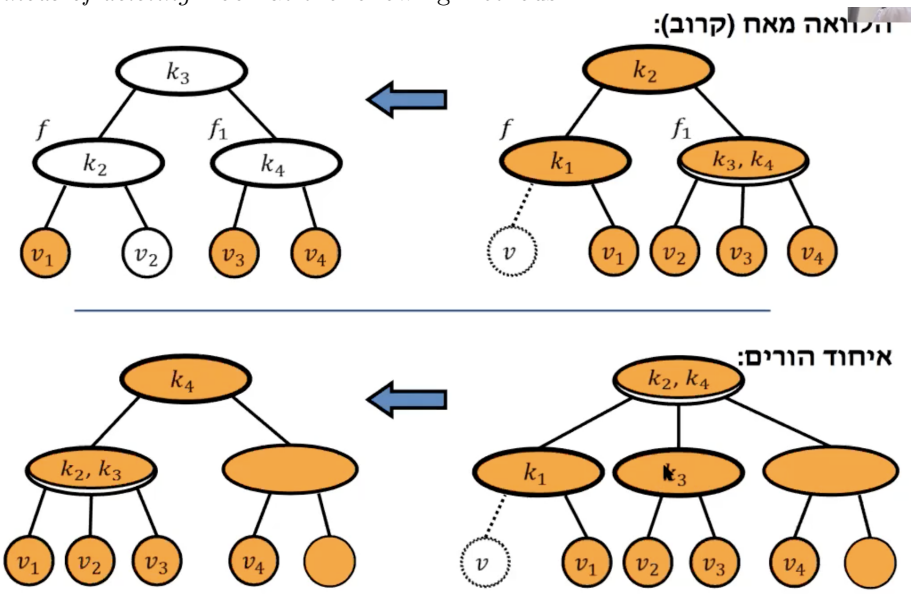
Deleting in $2-3$ trees. In deleting operation we want to do the following:

(1) Search the place of k key.
(2) Deleting the leaf which has k as key.
(3) Fixing the tree S.T for every node will have $2-3$ childs. the fix is made on the searching path of k from the leaf which was deleted to the root. in every level we will execute $O(1)$ operations.





Two methods of deleting. look at the following methods:



as we see above the first method is taking a brother child, first we were asked to delete 14 we search for it in the tree. now we see that 14 key which has childs with value 14 and we take it's child since it's brother has a 3 childs then we fix it's

parent key which is 14 in this case and change it to 15. now the second method is when there is no 3 childs, so we union keys instead in our case when we delete 3 we union $2, 7$ keys into one node and again we will get the same problem for the node which has key 3 and want to delete it, now we need to execute a fix to the tree by union keys again since 3 has now only one child which is $(2, 7)$ now we look at how many child it's brother (15) has in this case we have 2 childs so we union 3 node and 15 again and they will have 3 childs which are $(2, 7), 14, (16, 19)$ then we upadate the key indexes in the following way the left key will have the min key in the middle subtree and the right key will have the min key in the right subtree. now we have no need root which is 10 so we delete it.

Algorithm of deleting.

(1) search for k in the tree. if k doesn't exist, finish.
(2) let t be the value of the leaf which k it's value and let f be it's parent.
(3) delete t from the tree.
(4) $f \to v$.
(5) if v is a root and has a single child, then delete v and finish.
(6) if v still has two childs, finish.
(7) case A (taking from brother): if v has brother with three childs, then take one child from it's brother and finish. case B (parents union): if for all brother there is only two childs. then union v with a brother and update the key in the union node of keys, let f be the father of v.
(8) back to step 4.

Remark. notice that:

- we could put values in interior nodes and not only in the leafs.
- in real world we tend to generalize the construction for a bigger nodes and keep the data in the leaves.
- generalization: B trees.

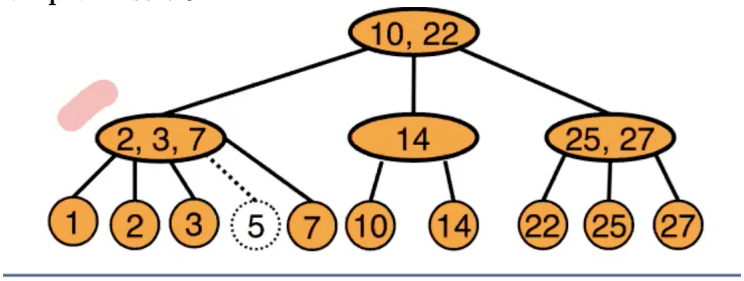
Generalization: B trees and $B+$ trees.

- in trees we saw, in every node there was between $2 - 3$ nodes. we want to generalize to every node between min and max childs.
- **Trees like that are called B trees (with a value in there interior childs) or $B+$ (which have values in the leaves).
- which condition we want to determine between min and max?
- important condition for split and union is $min \leq \left\lfloor \frac{max+1}{2} \right\rfloor$ otherwise we will have a non legal trees.
- E.g $(2, 3), (2, 3), (20, 40)$. since we know the $2 - 3$ are legal and $20, 40$ also since $20 = min \leq \left\lfloor \frac{max=40+1}{2} \right\rfloor = 20$.
- but not $(30, 33), (5, 7)$. since $5 = min \not\leq \left\lfloor \frac{max=7+1}{2} \right\rfloor = 4$.

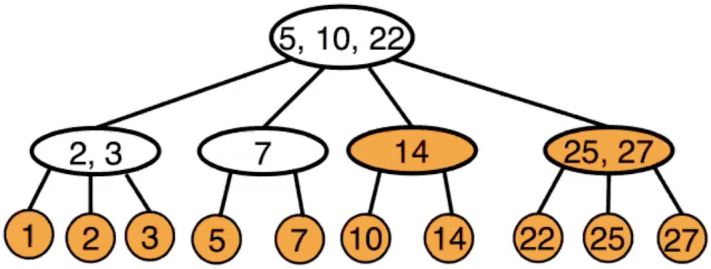
Definition. B^+ tree with rank m is a tree which satisfy the following:

(1) all the values exists in the leaves of the tree.
(2) for every interior node, expect the root, there is c childs when $\left\lfloor \frac{m}{2} \right\rfloor \leq c \leq m$.
(3) for the root number of childs $c$is: 0 (single node in the tree) or $2 \leq c \leq m$.
(4) for interior node with c childs there is $c - 1$ sorted indexes by there size.
(5) all the keys which exists in the subtree i are less then the i index and bigger equal the $i-1$. all the keys which exists in the most right subtree are bigger equal than the last index.

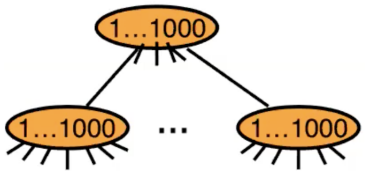
Example. insert 5:



in this case only one split required. half number of the nodes (rounded up) in the left subtree (and down rounded) in the right subtree.



Main use of $B+$: Documents and Database. the memory in most computers are made of two parts: (RAM) main memory, small but quick, or secondary memory (DISK, SSD) big but slow. the access time to a secondary memory is huge also: we read or write always in one ti,e block (E.g 1024 bits). documents and database are in disk and ofter organized in a **B+ trees** when the nodes are broad (block size) and the tree depth is very small.



$B+$ trees with hight rank are compatible here:

in every interior node we keep indexes and they are recalled from the dist as one block. in that way we get a pointer to the next next node which we need to call and it's also a block. the numbde of access to the secondary memory depend on the number of level (which is pretty low in this case). E.g if we hold at least 64 pointers in each block then in the second level we will have $64^3 = 4096$ childs and in the third $262,144$.

Compartion between AVL trees to 2-3 trees.

- asymptotic: both trees support dictionary operations in $O(logn)$.
- theoretically:the choosing between two of them is not neccessary.
- practically: $B+$ trees allow a wide node and low tree depth. which really adapt database and documents.

Invariant. examples we already saw:

- Invariant in a binary tree: every node bigger then the nodes in the left subtree and less than the node in the right subtree. which prove that: the time for searching, inserting and deleting is $O(h)$ when h is the tree height.
- AVL invariant: every node has a balance factor of $[-1, 0, 1]$ which show that $h = O(logn)$. if we also add the invariant of a binary search tree and some fixes in time $O(h)$ after inserting/deleting, we get that the searching time, inserting and deleting is $O(logn)$.
- 2-3 tree invariant: every leaf are in the same level, and every interior node has $2 - 3$ childs. which show that $h = O(logn)$, if we add the indexes correctnessin the nodes and we get that searching, inserting, deleting are executable in a $O(logn)$.

Not only update and searching.

- target: to add ability to the tree without harming complexity of the operations.
- E.g if we want to know if 129 is the median of the tree, we can copy all the tree to a array and check if 129 is in the middle.
  - but that very expensive.

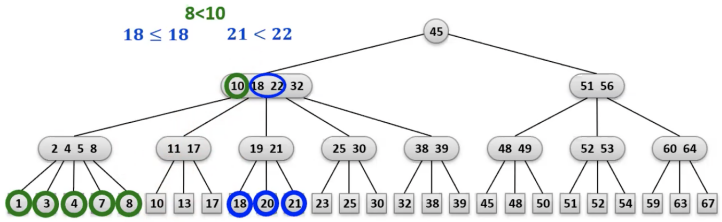
- can we doit in $O(logn)$ comlexity with affect the complexity of other operations?

B+ Trees exercises.

Definition. $B+$ trees with rank m is a tree which has the following properties:

(1) all the values are in the leaves. and all the leaves are in the same level.
(2) for every interior node expect the root there is c childs when $\lceil \frac{m}{2} \rceil \leq c \leq m$. (otherwise, for some operation which require split, we can get unlegal tree i.e with more childs than m). number of the childs to the root satisfy $2 \leq c \leq m$.
(3) for a node with c childs there is $c-1$ indexes $k_1, ..., k_{c-1}$which satisfy:
   (a) all the values which exist in the most left subtree onn the node are less than k_1.
   (b) all the values which exists in the subtree between the keys $k_{i-1}, k_i$are in the domain $[k_{i-1}, k_i)$
   (c) all the intevals which exeists in the most right subtree of the node are bigger or equal to k_{c-1},

Private case to a $B+$ tree is $2-3$ tree (rank 3).

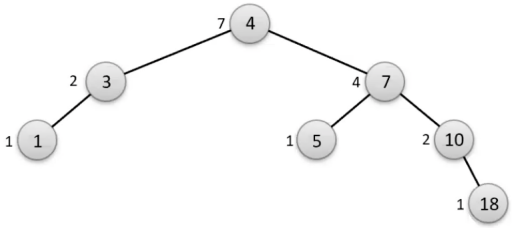


rank trees. index of element in a set of elements is the place of the element in the sorted sequence.

- for example, for the set $\{10, 1, 5, 18, 7, 3, 4\}$ the sorted sequece is $\{1, 3, 4, 5, 7, 10, 18\}$.
- from here $rank(1) = 1, rank(3) = 2$..
- notice that t
- he index start from 1. (and not from 0 as we used).

Definition. rank tree is a data structure which can execute $rank(x), delete(x), insert(x)$ in complexity of $O(logn)$ each one. rank tree is a balanced BST in which every node hold in addition to the existed data, a additional field w which keep the number of node is the subtree of the node.

Example. look at the following rank tree.



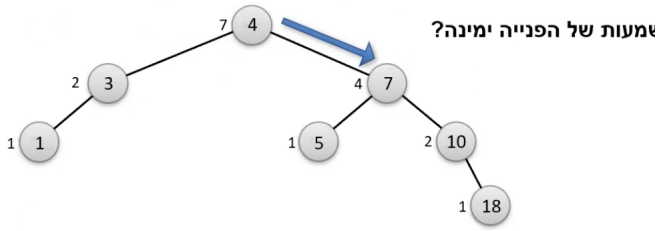
מבני נו

finding index in a rank tree algorithm. during finding x in the tree, we will count number of elements which are less than x in the tree.

- we intialize a helping variable $r \to 0$.
- while searching, everytime we go down right of a node c we will do $r+w(v \to left)+1 \to r$.
  - $w(null)$ is defined to be 0.
- when we arrive node x, we will execute $r + w(x \to left) + 1 \to r$.

Example. in the photo above:

- in the beginning $r \to 0$.

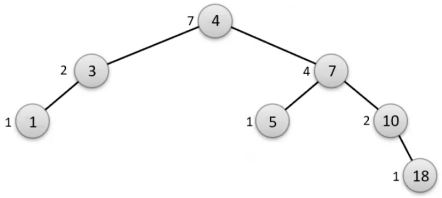


- we see that we go right since $5 > 4$ but notice that we go right from the root therefore, we update i.e $0 + \overbrace{2}^{w(v \to left)} + \overbrace{1}^{v} = 3 \to r$.
- then we search for x, and add it's index notice that from the node 7 we will go to it left child which is 5 so we don;t update r since we update only only with the node which we were go to it right childs in this example it was the root only which has rank of 3.
- so we return $r + w(null) + 1 + = 3 + 0 + 1 = 4$.

select(k) - *finding the element in the tree with index* k.. we will start from the root and execute the following algorithm.

- if $w(v \to left) = k - 1$, we will return the root. in this case we notice that all the number of keyd in the left subtree is exactly $k-1$ therefore, the root has index k and we return it. now we can do that recursively till we find the wanted index.
- if $w(v \to left) > k - 1$, we will search recursively in the left subtree the element which has k as index.
- if $w(v \to left) < k - 1$, we will search recursively in the right subtree the element which has $k - w(v \to left) - 1$ as index.

Example. find *select*(4) in the following tree.



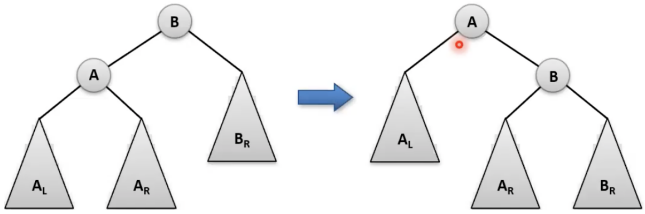
- we start from the root 4 we see that $w(3) = 2 < k - 1 = 3$ so which index we will look for and in which subtree?

- we see that in the left subtree+ root we have a three elements therefore, the index we are look for in the right subtree.. hence we will search for $k - w(3) - 1 = 1 \to k$ in the right subtree of the root 4.
- for node 7 we see that $w(5) = 1 > k - 1 = 0$ hence we search in the left subtree of 7 as the algorithm tell us but now with $k = 1$ and we see that $w(null) = 0 = k - 1 = 0$ therefore we return 5.

Algorithm to insert/remove from rank tree. for the end of the inserted\deleted node till the tree root, for every node v in the path:

- update $w(v) = w(v \to left) + w(v \to right) + 1$.
- go up.
- fixing the tree after inserting\seleting take $O(logn)$.
- did we finish? no, what about rotations update?

updating the new field after AVL rotations. we will illustrate that using LL rotation. the update for other rotations is identical. before the rotations the fields $w(\cdot)$ was true for all nodes.



update after the rotation:

- $w(A_r) + w(B_r) + 1 \to w(B)$.
- $w(A_L) + w(B) + 1 \to w(A)$. (notice that here we used the updated value of B).

the rotation operation still $O(1)$,hence the insertion\removing time from rank tree based on AVL tree is as insertion\removing time from AVL tree.

Some remarks.

- we can implement rakn tree on base of $2-3$ trees. the values $w(v)$ will relate only to the leaves.
- when we define rank trees which has other field, we need to execute the following steps:
  - determine in accurate what is the additional field which will be kept in every node.
  - how we are going to use those additional data to solve the given problem.
  - how we are going to hold the additional (after deleting\inserting) effeciently. E.g we saw that for standard rank tree based on AVL after insertion\deleting, we can update the tree S.T every node will hild number of nodes in it's subtree, without harming in complexity of $O(logn)$ for insertion\deleting.
- we can use rank trees to other calculations:
  - sometimes we are going to define specific variations of rank trees which adapt the problem. E,g for additional information we can save in node:
    - sum of keys in the subtree.
    - the minimal key in the subtree.

* etc..

Exercise. describe a data structure which will execute the following operations.
$insert(x)$ inserting a new element x in complexity of $O(logn)$.
$Delete(x)$ deleting a element x in complexity of $O(logn)$.
$Member(x)$ check if the element x in the structure with complexity of $O(logn)$.
$OddSum()$ find the sum of all elements with odd indexes in complexity of $O(1)$.

Example. which elements in the array, $OddSum()$ were return 6.

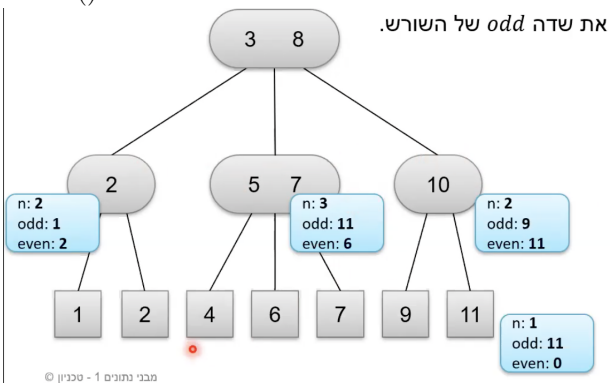
1	3	5	10

Solution. We will use a rank tree on a base tree $2-3$.

In every node v, expect the regular fields, we will keep the save the additional data.

- n - number of the leacrs in the subtree of v.
- odd - sum of leaves in the odd places in the subtree of the node.
- $even$ - sum of leaves in the even places in the subtree of the node.

$OddSum()$ we return the field odd of the root.



notice that we need to rake the sum of odd indexes i.w in this case we need to take $1+4+7+11=23$ now we see that in the first we have 2 i.e we got two indexes we we take the even odd sum, now we know that next the middle subtree start with odd index 3 in this case so we also take the odd sum which is 11 then we get into the right subtree we know that it starts with a even index which is 6 so we take then even sum which sum the elements in the odd indexes i.e 11 therefore, we get in total $1+11+11=23$ we will se how we are going to formalize it in insert operation,

$Insert(x)$:

- we will insert x to the $2-3$ tree when in the node which represent x in the low level on the tree the fields values are:

$$1 \to n, x \to odd, 0 \to even$$

- notice that the only node in which the values of $even, odd, n$ changed are only the nodes on the path from x to the root. hence, there is no need to update the nodes which don't exist on the searching path of x.
- we will walk on the searching path of x from down to up. in each node in the path, we will upadte it's $even, odd, n$ values

$$n(left)+n(middle)+n(right) \to n$$

now notice that we need to split into cases,

$$A+B+C \to odd$$

$$odd(left) \to A$$

$$\left\{ \begin{array}{cc} odd(middle) & n(left) \text{ even} \\ even(middle) & otherwise \end{array} \right\} \to B$$

notice in case above it was even therefore, we took the odd middle which was 11.

$$\left\{ \begin{array}{cc} odd(right) & n(left)+n(middle) \text{ even} \\ even(right) & otherwise \end{array} \right\} \to C$$

now we will calcualte even as the same of odd i.e

$$A+B+C \to even$$

$$even(left) \to A$$

$$\left\{ \begin{array}{cc} even(middle) & n(left) \text{ even} \\ odd(middle) & otherwise \end{array} \right\} \to B$$

$$\left\{ \begin{array}{cc} even(right) & n(left)+n(middle) \text{ even} \\ odd(right) & otherwise \end{array} \right\} \to C$$

we update after delete operation as it's in insert. i.e we will update the nodes in the searching path. every update on the fields $odd, even$ after insretion or deletion which require $O(logn)$ complexity, since we execute constant number of operation to every node in the searching path, and the tree height is $O(logn)$, the complexity time for every update is $O(logn)$.

Strings.

Definition. string is a sequence on the alphabet some $\sum$. we will assume that it's bounded. every string always end with unique label \$ (we will assume $\$ \notin \sum$) which indicate the end of the string. the label \$ is lexicographic less tahn all the label in $\sum$.

Exercise. implement a data structure which support the following opertaion on $\sum$.

$insert(s)$ - inserrting a string to the structure.

$delete(s)$ - deleting a string from the structure.

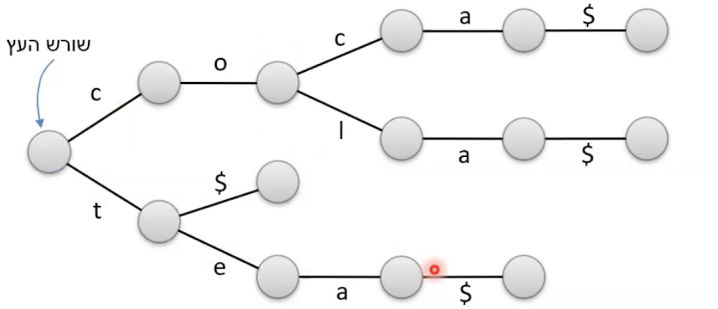
$find(s)$- is the string exists in the structure.

we wish to find a data structure which is time complexity doesn't depend on the time of each operation doesn't rely on the number of strings, but on it's length only.

implementation using a data structure which we alreadtysaw. assuming we have in the structure n strings and each of them with length m.

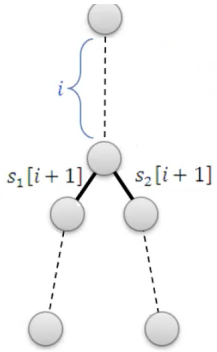
we will hold a balanced tree (AVL, 2-3) in which every node hold string. delete, find, insert will be executed in a $O(logn)$ but each comparsion operation between strings will take $O(m)$ time complexity. therefore, each of the following operation will take $O(mlogn)$. this is bad implementation sinceit rely on the number of strings.

Trie. we will hold a tree in which every interior node there is at most $|\sum| + 1$ childs. the arches to the childs every node will save as a array in that length. every arche is marked by the label which match it.



every string is represented by the path from the tree root to the leave.
the operations $find(s), insert(s), delete(s)$ take $O(|s|)$ time complexity.

Tranverssal in a trie. a preorder tranversal on a trie, we walk on all the arches of each node Lexicographically, we reach to a leave which adapt a string s_1 before it reach to leave which adapt a string s_2 if $s_1 < s_2$. i.e if we work with stack or linked list we can use the tranversal to print all the string in lexicographic order.



E.g if we have two string $abcd, bcde$ we first print $abcd$ since it's lexicographically bigger than $abcd$ so in the tranversal we reach it's leaf after.

Remark. tranversal inorder not defined on a tries which are not binary. sice, we first print what we have in the root in which we two end of different strings split, and we can't print it after we print it's childs therefore, inorder is not defined on trie trees.

rank trees. in a lot of uses if searching trees it's reccomended to keep other data in every node in order to make operations faster. we will give some examples.

Definition. the index *rank* of number x in the set S is it's place in the sequence sorted of S elements. E.g element 5 in the set $\{8, 3, 7, 5\}$ has rank 2.

Problem. implement a searching tree which support, in addition to the operations inserting/searching/deleting, also finding the index of x in tree with $O(h)$ time when h is the tree height.

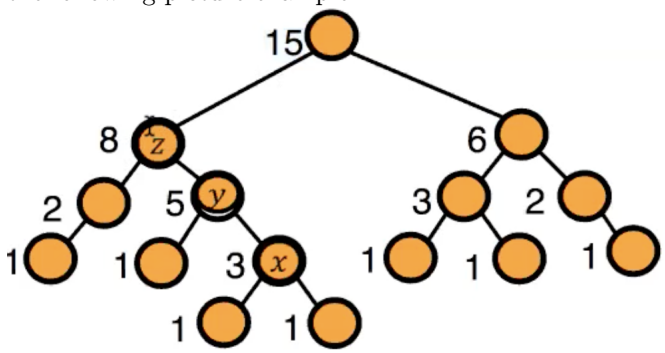
Problem. implement a searching tree which support the operation of calculating a elements sum in tree which are less than x with $O(h)$ complexity.

- in those problem there is need to keep additional data in the searching tree in order to implement the other operations with the complexity required.
- we will see what is the data we are going to keep and how are we going to use it.
- after that we will see how are we going to update the additional data while inserting and deleting.

Definition. a rank tree is a tree in which in every node v we will keep the number of nodes in the subtree which s root is v.

Solution. for problem 1.

we will use a rank tre, denote the rank by $n(v)$ appeat in the left of the nodes in the following picture example.



let *path* be the searching path of x. every node which is less than v or less equal than x in the path we will count v and all the nodes in the left subtree of v. we can calculate it as following:

$$rank(x) = |\{z, y, x\}| + 2 + 1 + 1 = 7$$

$$rank(x) = \sum_{v \in path, v \leq x} (1 + n(v.left))$$

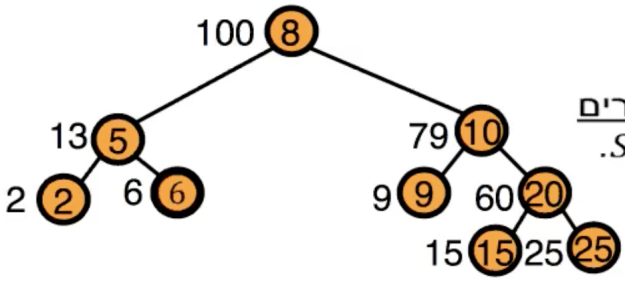
Corollary. *in a rank tree finding the rank take $O(h)$ time complexity.*

Example. a rank tree is a example of a data structure in which in every node will save the data about the subtree below it. type of data depend on the problem which we want to solve.

tree with sum.

Solution. for problem 2.

in every node v we will save the sum of element of the subtree whch is it's root is v. we will denote this number by $S(v)$.



let *path* be the searching path of x. to every node v which is less or equal tham x in the path we will add the value of v and all the nodes in the left subtree of v. using the fields $S(v)$ we coulds calculate it as following:

$$SumTo(x) = \sum_{v \in path, v \leq x} (v + S(v.left))$$

Corollary. *the running time is $O(h)$. how are we going to insure a running time of $O(logn)$ i.e we need to insure that our tree height is $O(logn)$.*

Problem. implement a searching tree which support finding a rank of element x in a $O(logn)$ time.

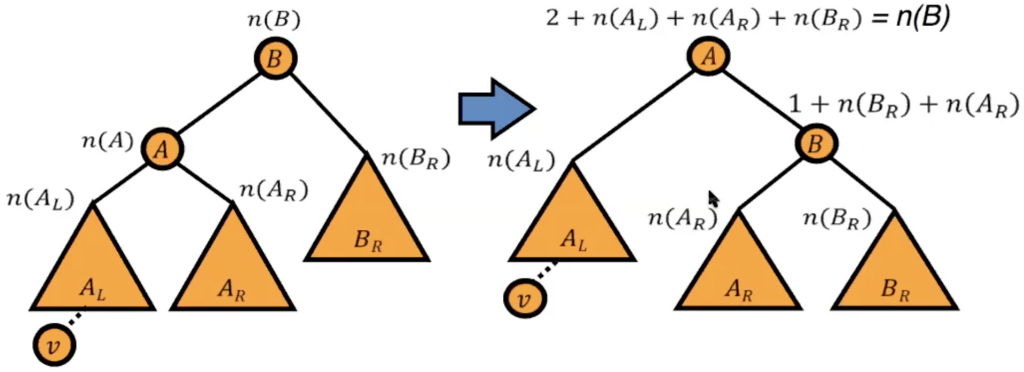
Solution. E.g, using a AVL tree. in which in every node v we will keep the number of nodes in the subtree which is root is v. remember that we denote it by $n(v)$. we will update $n(v)$ in inserting and deleting operation and rotations as well.

update the additional field.

how are we going to update $n(v)$ - number of nodes in the subtree
- in every subtree of AVL?

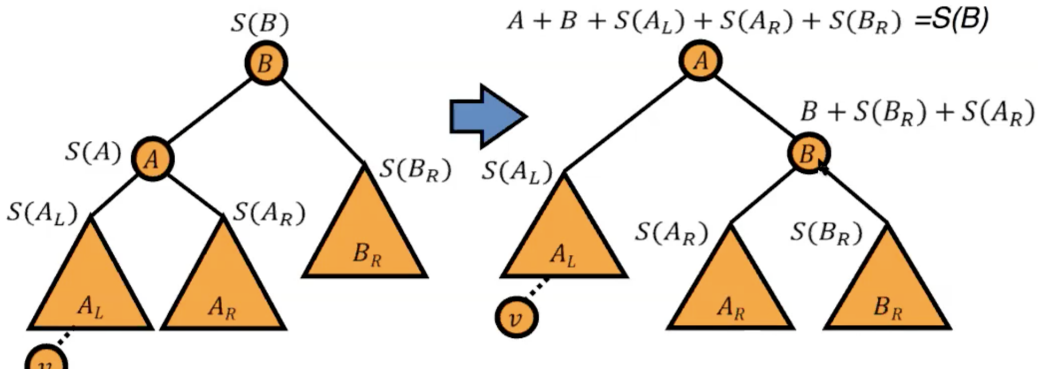
(1) In inserting operation we will increment $n(v)$ by 1 for the path length from the root the leaf which were inserted. In deleting operation we will increment $n(v)$ by -1 for the path length from the root the leaf which is going to be deleted.
(2) In rotation time we will update the new field as required, while using the fact that $n(v)$ depend only $n(v)$ values of the subtree, which already were fixed.

Example. E.g in LL rotation we will update like this:



other example for upadating the field. how are we going to update $S(v)$ - the sum of keys in the subtree - in every subtree of AVL tree.

in identical way of updating $n(v)$ previously - while inserting/deleting we will update the sum from the path which start from the root to the leaf which is inserted/deleted. in rotation time we will update the root field as required.



Trie - Dictionary for strings. The implementatiton using tree. for every node there is at most number of childs as size of alphapet + 1. every edge is marked by char, the char \$ (which is not in $\sum$) mark a end of string. we will consider that (wanted case) in which size of $\sum$ is contsant.

Example. Trie for the following string will look as following ac, a, bc when end char if string is \$ is the most small exicographically.

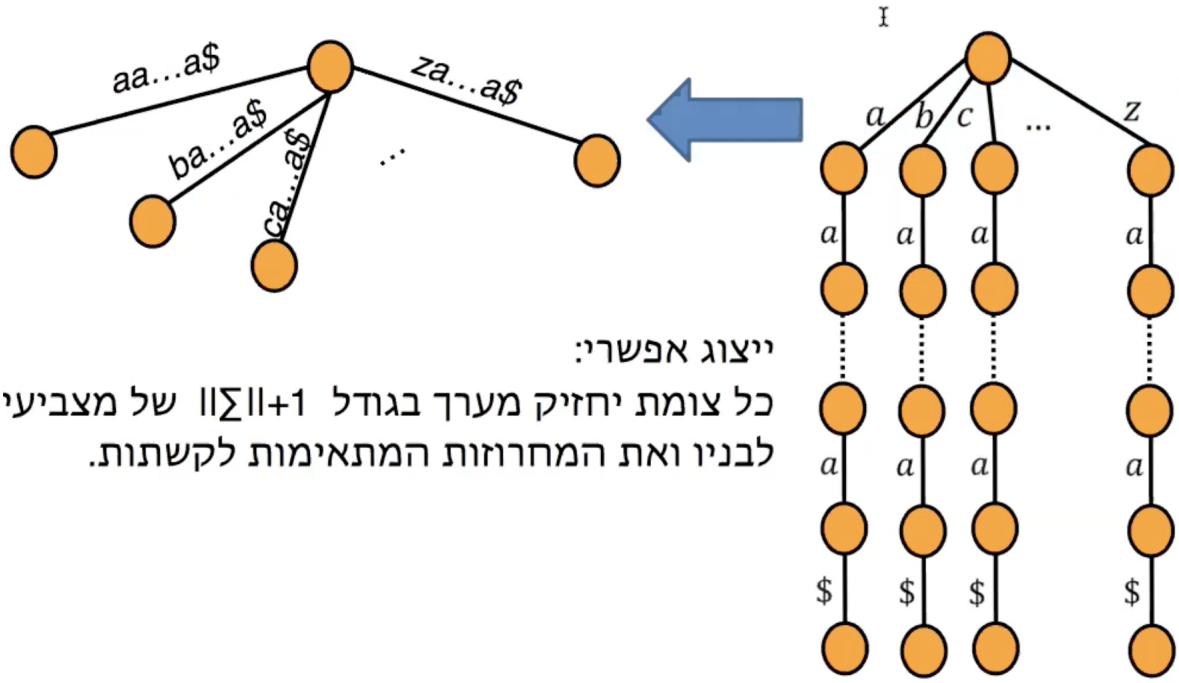
Remark. every string in the structure define a path from the root to the leaf. all the operations are executed by a check for the required path.

Implementation complexity. The time required to execute is determined by the length of the string $O(|s|)$ and not by the strings number n. Moreover, the space is complexity is linear as sum of strings length in the structure.

Lexicographic order of strings. Given two strings s_1, s_2 and t first letter are identical.

- The order is determined by the letter in the place $t + 1$.
- If one of the strings with length t and the second not then the string which has less length is the smallest.
- If two of them are with length t then the strings are equal.

compress trie. We will omit from the tree nodes with one childs by changing chain of edges in one edge which will called by the encoding label of the following string.



Possible represnetation:

every node will hold a array with size $||\sum||+1$ of pointers to it's childs and the wanted string to the arches.

Remark. Notice that we execute compress only in the nodes which don't have path split E.g assuming that in node z we had other string $bbbbb\$$ after z so node z will be splitted and we can't compress now as one string $zaa\$$ therefore, we compress only $aaa\$$ and $bbbb\$$ and the node z will still the same i.e we compress only the pathes of the children of z.

Tries compress, the way iy effect number of nodes. Let M be the number of nodes intree and n number of leaves. during compress for every of $M-n$ interiod nodes there is at least two childs.

Lemma. *In every tree which has a t least two intrior childs, number of vertices satifsfy* $M \leq 2n-1$.

Proof. Since we are talking about tree, and since for every interior node there is at least 2 childs, satisfied that:

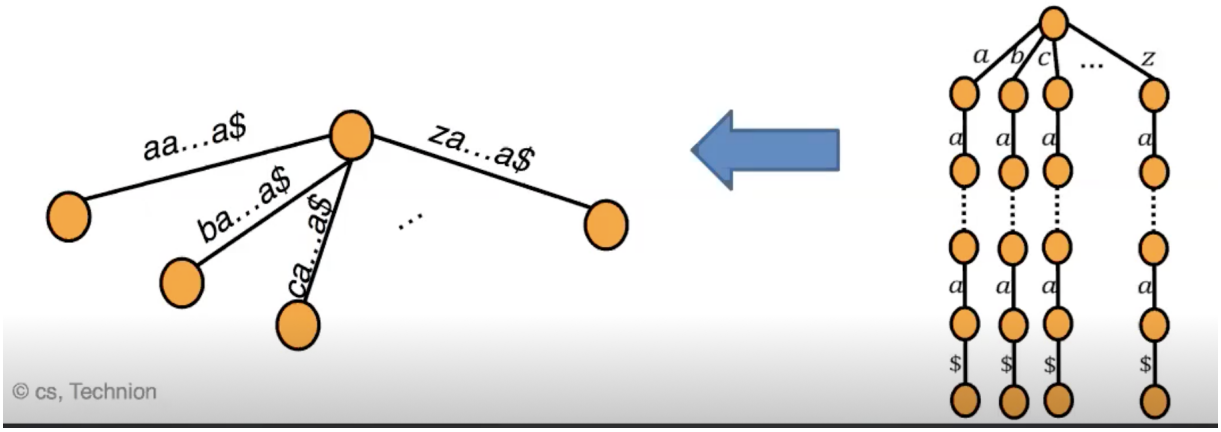
$$2(M-n) \leq_* \sum_v d_{out}(v) =_{**} M-1$$

$$\Rightarrow M \leq 2n-1$$

explain $*$: *given that every of the* $(M-n)$ *interior nodes there is at least* 2 *childs.*

explain $**$: *in a tree number of arches is less by* 1 *from number of nodes.* □

example.



Corollary. *After compressing Trie the number of nodes is linear in number of leaves (i.e number of strings) in Trie. the saving is important in advance when the label of arches would be encoded in compactive way.*

Skip Lists

- Balanced trees: they implmement dictionarty opertsiond as (searching, inserting, deleting) with time complexity $O(logn)$ in the worst case.
  - But implementing the operations are not trivial (preserving the balance).
- Regular trees are simple but don't insure balance.
- Skip lists - simple to implement, but the insure on operations complexity is average: $O(logn)$ in average.
  - but, the probability that the execution time exceed a lot (3 times) from the average time is negligible (less than 1 out of $100,000,000$).

What is "average"?

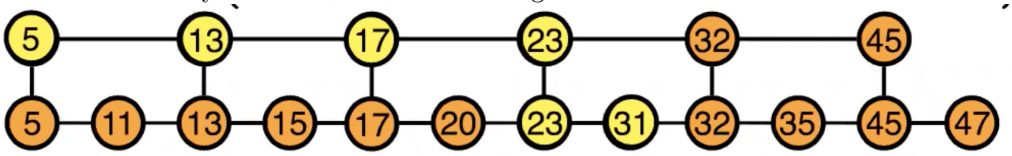
- Average on input: E.g, we saw previously the in unbalanced tree that tree "behave well" if the input order is random of keys.
  - it's a weak term. since the order is controlled by enemy (adversary) which use the system operations. i.e a user can enter 1 2 3... in sorted way then we get lace without knowing that he will get bad tree.
- notice that the assumption we did is on the input, and for n elements there is $n!$, the average of the $n!$ trees is $O(logn)$. but- notice that the assumption was on the input.

- the problem in this analyze it that appearing a permuatation in the input is not known, and there is no reason to assume that it's uniform.
- in skip lists, which we are going to see, we will get executing time of $O(logn)$in average. moreover, the structure and the operations we do depend on the lotteries which the computer do and not in the order of the inserting.

- the average is calculated by the lotteries and not by the probability assumption in the input, algorithm like that is called random algorithm.

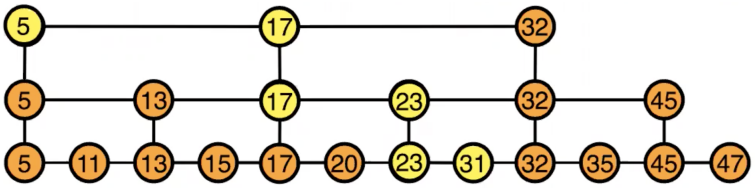
The main idea os skip lists. searching for element in a list take a expensive time inserting/deleting/searching $O(n)$.

as we in the picture above, if we want to seach for 31 then in the worst case we need to visit all the n nodes. now in order to make the search more faster we will add a sublink of every second element as following:



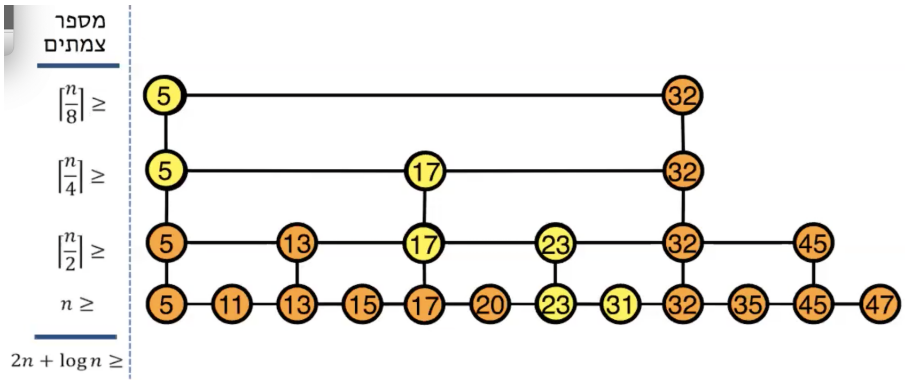
now we do the search as following:

first we go into the sublinl first node which is 5 in this case the we see that $5 < 31$ which is the element we wish to find, then we go on to the next element in the sublink which is 13 and we see also the is less, we continue till we reach 32 in the sublink and $32 > 31$ therefore we get down to the most low level which is the origin link and start searching from 23 which is the previous element of 32 in the sublink then after one step we find 31. notice that we save a lot of time and we can do it more and more, i.e we can add subsublist to the sublist and keep doing that till we reach the *logn* level which is the most higher level we can get.



Space complexity of a skip lists. notice that if the number of elements is n then the number of elements is in the second level is a $\lceil \frac{n}{2} \rceil$ and in the next level is $\lceil \frac{n}{4} \rceil$ and go on. now notice that we have $\frac{n}{2^i} \leq 1 \rightarrow i \leq logn$ which is the higher level and it will have 1 element therefore our sum will be as following:

$$n + \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{n}{4} \right\rceil + ... + 1 = n + (\frac{n}{2} + 1) + (\frac{n}{4} + 1) + ... \leq 2n + logn = O(n)$$



Illustrating photo.

complexity of searching in skip lists. we will look at the searching path from the ending node to the starting node. in the transition from one level to other we pass on two pointers at most, therefore, the longest path is at most $2\lceil logn \rceil = O(logn)$.

Problem. It's hard to keep accurate strcuture while inserting/deleting with out oragnizing all the levels again, (inserting\deleting let us update all the structure after). E.g if we want to add 16 to the graph above we see that the order will be not good since we determined that every between to node we duplicate, hence, we are required to change the structure in the second level.

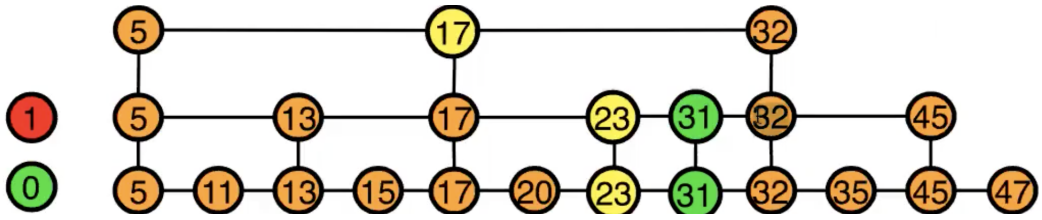
Solution. rolling a coin. then we insure that in every insert/deleting we add randomly and there is no unique structue i.e dupicating every second node in the link as above, then when we delete or insert our struture is random and it won't damage anything. hence,

- In inserting: we will roll the coin for the number of levels to insert the node, every level up with probability of $\frac{1}{2}$.
- In deleting: we will delete the key from all the levels.

Insert using rolling coin. Algorithm for inserting:

- Insert a key k if it exist, finish.
- hold a pointer to the most right node in the searching path.
- Add a new node to the lowest level nad determine the key to be k.
- According to the order levels from down to up:
  - roll a coin $-toss()$
  - if it give 0 then add a new node to the level above current level, and let the key be k, if in the high level 0 were rolled, add other level.
  - if 1 were rolled then stop.

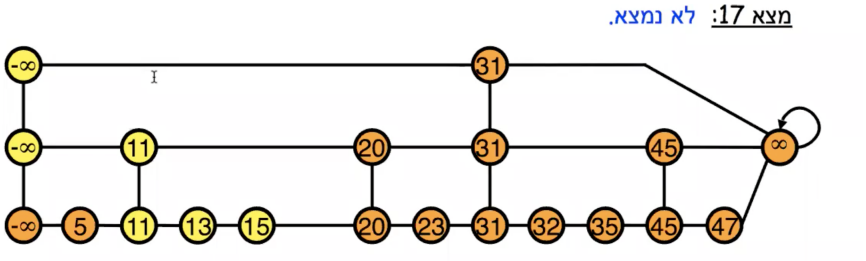
Example. look at the following graph:



notice that when we inserter 31 in the first roll we get 0 then we add to node to the level above with key 31, then we continue and roll again, then we got 1 in the second roll and we stop adding new nodes to levels above with key 31.

Remark. Some remarks for implementing a skip lists.

- as in linked lists, it's easy to add in each level one node in the beginning of the list with key $-\infty$ and one node in the end with key ∞. it help us when we delete avoiding a private cases when we delete.



Space analyze.

- In every level the algorithm roll a coin in order to determine whether to duplicate or not.
- the expectation of level number which it's key exist into is equal to the expectation of number of times which we roll till we get 1.
- the following expectation is 2. (we will see).

Corollary. *The expectation of number of nodes in a skip list is* $2n$*,when* n *is the number of keys in the structure (the expectations of numer of node to each key is* 2 *and there is* n *keys) from here, the space expectation required is* $O(n)$.

Rolling coin. Assume that a coin has probability p to get 0 and $1-p$ to get 1 so what is T, the expectation time in which we need to roll the coin till we get 1.

Denote Q_i the probability to get 1 the first time in the roll i (the follwoing distribution is called geometric distribution) moreover, satisied that:

$$Q_i = p^{i-1}(1-p)$$

$$T = \sum_{i=1}^{\infty} i \cdot Q_i = \sum_{i=1}^{\infty} i \cdot p^{i-1}(1-p) = (1-p)\sum_{i=1}^{\infty} i \cdot p^{i-1} =_* \frac{1-p}{(1-p)^2}, |p| < 1$$

when in $*$ we used the fact that:

$$\sum_{i=1}^{\infty} i \cdot p^{i-1} = (\sum_{i=1}^{\infty} p^i)' = (\frac{p}{1-p})'$$

i.e the expectation of number of roll required till we get 1 is:

$$T = \frac{1}{1-p}$$

and for a fair coin $p = \frac{1}{2}$, hence $T = 2$ required in average.

Theorem. L *is the expectation of length of a searching path in a skip lists with* n *keys, satisfy that:*

$$L \leq 2log_2(n) + 2$$

Proof. we will look on the searching path from the end node to the start node that path is made of a up move and left move. now notice in order to reach the node we wish er can go up or left (since we start from the end of the lowest level E.g in the photo above we start from 47) and each has probability of $\frac{1}{2}$ now if we go left. now each time we do one step we have probabiliy of $\frac{1}{2}$ from left or up now if we go left then we do one step + still k levels till we reach the node and if we do step up then we do one step + $k-1$ levels so we can build a recursive formual. denote by $c(k)$ the average path length which go up k levels. satisfied that:

$$c(0) = 0$$

$$c(k) \leq \frac{1}{2}(1 + c(k)) + \frac{1}{2}(1 + c(k-1))$$

explain: step to the left in probability of $\frac{1}{2}$ and step up in probability of $\frac{1}{2}$.
hence we got:

$$c(k) \leq 2 + c(k-1)$$

$$c(k) \leq 4 + c(k-2) \leq .... \leq 2k + c(0) = 2k$$

i.e the length of a average path from level 1 to the highest level is at most $2(M-1)$ when M is the number of levels.

Remark. notice that here we look at specefic path which start from the node to the beggining node which must take the most time expectation and it's suffcient expectation for all possible pathes since it's the longest.

now, number of level M is the distubute as the max of n geometric random varibles when each element in the link represent a random variable distribute geometric since it like rolling a coin and we can show that it's less than $log_2(n) + 2$. E.g in the example above $M = 4$ and in general it's bounded by $log_2(n) + 2$ (we will not show) hence, the expectation of a searching length is at most:

$$2(log_2(n) + 2 - 1) = 2log_2(n) + 2$$

□

time complexity:

- Worst case: what is the worst time which we can get?
- Averaage time on a input: given input from such a distribution, how much time operation would take in average?
- Average time on rolling a coin: in every run we will roll coins which our calculation will consider and we ask: how much time take a operation in average on the space or rolling the coins.
- amortized time: after we rum m operation, what is the average time in which every single operation would take, in the worst case? (not random).

Time and amortized cost.

Definition. if m of operations are executed in total time of $T(m)$ then the amortized time for every operation is defined to be $\frac{T(m)}{m}$.

importance. for a log sequence of operations ofter it's important to us how many times all of the cost and not only single operation. if the majority of the operations are fast and small part are slow, then the effecieny op the slow operations devides by the sequence of all operations.

formal. if we execute m operations which cost $c_1, ..., c_m$ respectively, then we want upper bounder for $\frac{1}{m}\sum_{i=1}^{m} c_i$. this is different from what we did till now which is bounding the most expensive operation i.e $Max_{i=1}^{m}\{c_i\}$. we will see three different method: aggregate analysis, potential, accounting.

Stack with a expanded deleting operation. remember the regular operations of a stack $push(S,x), push(S)$ which take $O(1)$. we will define the cost to be one unit. we will define a additional operation.

$multipop(S,k)$: remove from the stack the k upper elements, if there is less than k element then empty the stack.

implementation. by applying operations $pop(S)$ in loop till the stack is empty of removing the k elements. the time $O(min(s,k))$ when s is the number of elements in the stack while executing the operation, the codt of operation is defined as the number of operating $pop(S)$.

Aggregate Analysis.

- In aggregate analysis in order to evaluate amortized analysis we show directly that every sequence of m operations cost in the worst case $T(m)$,and from here the amortized cost of operation in the worst case is $\frac{T(m)}{m}$.
- In analysis of worst case - a single operation could cost m hence, in the worst case for a sequence of m operations it require cost of $O(m^2)$.
- By aggregate analysis method: for every sequence of m operations required in the worst case cost of $O(m)$, when indeed every element removed were needed to be inserted. from gere when we start with empty stack the total cost of m opertions is $O(m)$ i.e the amortized cost of every operation is $O(1)$. this result is for the worst case. it's true for every sequence of operations with length m. E.g if we insert 10 element to the stack each insert will take time of $O(1)$ now notice that we executed 10 operation then we execute *multipop* which take also 10 therefore, the avarage amortized time in this case is $\frac{20}{11} \leq 2$ i.e in average every operation take less than 2. notice that this is way realistic than the worst case method in which we assume that every m operation take m^2 complexity i.e it assume that all the operation are expensive which is not rational.
- the idea is that in general is in sequence of operations can't be that all the operations are expensive.
- notice that we are looking to a sequence of operation which is mix of all stach operation pop, push, multipop. and in the worst case we assume that we always have multipop which is not the case in general thays why aggregate analysis method is more convincing.

Accounting method. In the following method we debit rach operation a cost which is called the amortized cost a_i. in cases in which the amortized cost of operation is bigger from the real cost c_i. the differece will give us credit. and we can use this credit in future to paymentm assistance to the operations in which

there aamortized cost is less than there cost in real. if we don't get debt, since in total the amortized costs constitue a upper bounder for the cost of operations in real i.e

$$\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} c_i$$

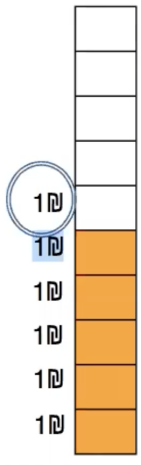
Example. The actual osr of stack operation is:

- 1 shekel for *pop*.
- 1 shekel for $push(x)$.
- $min(k, s)$ shekel for $multipop(k)$ operation.

We will define a amortized cost of 2 for every $push(x)$ and 0 to the other operations.

well, how we insure we don't get into debt?

for every $push(x)$ operation we pay 1 shekel and we deposit 1 shekel for furure operation, which will remove this element from the stack. while removing we give the one shekel which we have as credit.



E.g in the photo above assuming that we do push 5 times we will have 6 as credit, since every push take in real 1 shekel by in the amrotized cost we define it as 2 so we will have 6 as credit, now after that we operate *pop* (which cost 0 in amrotized cost), so we can see that we use the 1 shekel from the credit and we stay with credit of 5, afterward we do $multipop(5)$ and we use 5 elements but we have that it cost in the amrotized cost 0 so we can pay using our credit which is 5 it's real cost.

Since, we don't get into a debt we get:

$$2m \geq \sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} c_i$$

hence, amrotized cost for operation is $O(1)$.

The potential method. We will consider a paymeny in advance for expensive operations as a potential energy of a data structure as a one body. potential in which we can free in order to pay for the future operations. we will denote that data structure after operation i by D_i any ϕ is the potential function. in this

method we will not determine in advance the amortized payment a_i. instead, the amortized payment will be determined by the cost in real c_i in addition of a potential difference:

$$a_i = c_i + \phi(D_i) - \phi(D_{i-1})$$

In total all the operations get:

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} c_i + \phi(D_m) - \phi(D_0)$$

and in all sequence of operations the final potential is not less than the intial potential i.e $\phi(D_m) - \phi(D_0) \geq 0$ we get that the amortized cost which include the sequence of operations will be upper bounder on the real cost of the sequence.

Example. In the example above we will define ϕ to be the number of elements in the stack, the potential is 0 in the beginning and $\forall i$ it's positive $\phi(D_i) \geq 0 = \phi(D_0)$.

In the following definition the amortized cost of operation is:

- for $push(x)$: $a_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + 1 = 2$.
- now for *pop* notice that we have have real incresing in the by 1 but the potencial decrease by -1 therefore we get 0 as amortized cost.
- for $multipop(S, k)$: we have,

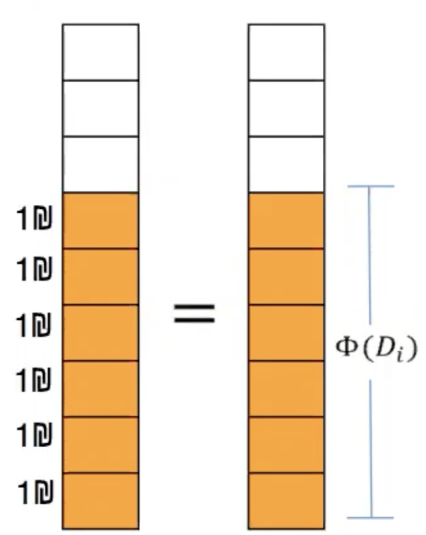
$$a_i = c_i + \phi(D_i) - \phi(D_{i-1}) = min(s, k) - min(s, k) = 0$$

Corollary. *we conclude that on all operations we heve:*

$$2m \geq \sum_{i=1}^{m} a_i = \sum_{i=1}^{m} c_i + \phi(D_m) - \phi(D_0) \geq \sum_{i=1}^{m} c_i$$

i.e the amortized cost for m operations is $O(m)$ and it's a bound for the cost in real.

Remark. notice that in the accounted method and in the potential method for the *multipop* proel were identical i.e for $push(x)$ we had 2 and 0 for the other operations, was it random?



Example. The binary counter problem.

we will check the three methods by operations on a binary counter with k bits. the representation so the numberator is $A = a_{k-2}a_{k-2}.....a_0$ and it's value is the repesented number in binary base. when a_0 is the LSB. the numberator start from 0 the operation which we want to support is increment operation which inccrease by 1 the numbrator value.

$$A = 00000000$$

$$A = 00000001$$

$$A = 00000010$$

$$A = 00000011$$

$$A = 00000100$$

$$\dots$$

$$A = 11111111$$

$$A = 00000000$$

$$A = 00000001$$

The operation is implemented in the following way:

```
increment(A){
        i ← 0
        while (i < length(A) and a_i == 1){
                a_i = 0;
                i = i + 1;
        }
        if (i < length(A))
                a_i = 1;
}
```

notice that this code increment E.g look at

$$A = 00100111$$

now after we we do *increment*(A) first we intialize i to 0 then we check we don't exceed the length of A and we notice that first we get into the while loop we can see that the while stop when we see $a_i = 0, i = 3$ then we convert all the the three bits to 0 afterward we check if i after the loop still less tham length of A if it less then we but in $a_i = 0$ so we get:

$$A = 00101000$$

worst case analyze.

The cost of the increment operation is linear depend on the number of bits which changed there value. since there is k bits, in every operation there is at most k bits which will change it's value, hence the cost of sequence of m operations is bouned by $O(mk)$. hence, a simple bounder for cost and amortized time for operation is $O(k)$. we will see that this analuze is not good.

aggregate analysis. we will look first at a sequence of *increment* operations:

$$A = 00000000$$

$$A = 00000001$$

$$A = 00000010$$

$$A = 00000011$$

$$A = 00000100$$

We can notice that each bit change it's value in every call to increment. moreover:

- a_0 bit change it's value in every call to increment.
- a_1 bit change it's value in every two calls to increment.
- a_2 bit change it's value in every fourth call to increment.
- $\cdots$
- a_i bit change it's value in every 2^i calls to increment.

therefore, the total cost is bounded by:

$$T(m) \leq \sum_{i=0}^{k} \frac{m}{2^i} \leq \sum_{i=0}^{\infty} \frac{m}{2^i} \leq 2m = O(m)$$

in other word, the amortized cost for operatin is $\frac{O(m)}{m} = O(1)$.

accounting method. we will define a amortized price of 2 shekels to determine value of bit as 1 and 0 shekels to determine value of bit as 0.

idea: in order not to get into debt, when the value of bit is determined to be 1 *we pay* 1 *from* 2 *in order to convert the bit, the other shekel will serve us as a credit for a converting to* 0 *operations on this bit. e.g if we took* $A = 101011001$ *and* $m = 3$ *first we pay* 2 *shekels (amortized money) since we change it to* 101011010 *we pay* 0 *for the first converted bit and* 2 *in order to convert seond bit to* 1 *in this case we are not in debt, now we continue with* $A = 101011010$ *notice that it has to in the first place* 0 *then we pay* 2 *(amortized price) in order to convert it to* 1 *but indeed we pay only one since we change one bit only but we will have* 1 *as credit then we get* 101011011 *afterward, we see that* $m = 3$ *now and we see that we have* 0 *in the* 3 *bit in this case we change first two bit to* 0 *and the third to* 1 *when we have* 1 *as credit from previous operation, now change two bit of* 1 *in real take two but amortized take* 0 *and chinging third bit from* 0 *to* 1 *take* 2 *amortized therfore, we have cost of* 3 *in real and amortized we have* 2 *to pay and one credit therefore in total we have* 0 *debit at then end. since, we can never get into debt stem that the amorized price which include operations of increment is* $2m = O(m)$ *(since every increment change only one bit to* 1 *and the rest to* 0 *which cost* 1 *on out amortized price). and it's upper bounder for the real cost. i.e the amortized cost for operation is* $O(1)$..

potential method analyze. let b_i be number of the 1 in the numerator we will define the potential after i operations of increment, to be 1 in the numerator i.e $\phi(D_i) = b_i$. the intiuition is the a decrease in the cost of 1 while doing operation which two many bits convert will pay for the most of the operation. we will see that indeed it work. Calculating the amortized cost of i increment operations.

- denote by z_i the number of bits which covert to 0 in the i operation.
- number of bits which convert is at most $z_i + 1$ hence, the cost in real of operation i imply $c_i \leq z_i + 1$.
- number of 1 in the numerator after operation i , b_i satisfy:

  $$b_i \leq b_{i-1} - z_i + 1$$

  e.g if we take 0011 and increment we get 0100 not $b_i = 1, b_{i-1} = 2, z_i = 2 \to 1 - 2 \leq -2 + 1$

  Example. look at:

  $$A = \cdots 1010110\ \underbrace{11111111}$$

  $$A = \cdots 1010111\ \underbrace{\overbrace{00000000}^{z_i}}$$

summary. if z_i is number of bits which get 0 in the operation i then:

- the cost in real satisfy that: $c_i \leq z_i + 1$.
- the potential difference satisfy:

$$\phi(D_i) - \phi(D_{i-1}) = b_i - b_{i-1} \leq 1 - z_i$$

Corollary. *the amortized cost is constant:*

$$a_i = c_i + \phi(D_i) - \phi(D_{i-1}) = (z_i + 1) + (1 - z_i) = 2$$

hence, we see that the first potintial is 0 and after i operatins for all i it's positive. therefore, the amortized price which include m operations of increment constitue a upper bounder on the total cost in real. this bounder is $2m$ hence the cost of amrtized operation is $O(1)$.

Remark. notice that the intituition the the i should pay for the operation which convert from 1 to 0 therefore, we can see that if we translate it to equation the increase of 1 from number to other in operation i is $b_i - b_{i-1}$ and it's cover the conversion to 0 of the bit in opertion i which is at most $z_i + 1$.

Amortized complexity exercises. Assuming we have m operations (identical) in the data structure, here is the time for executing each operation.

$$\overbrace{O(1), O(1), O(1), ..., O(m)}^{t_1 \qquad t_2 \qquad t_3 \qquad t_m}$$

Definition. Amrtized complexity, let F be a set of operations. we will say that F run in amortized complexity of $O(g(n))$ if every sequence with length m of operation from set F run in complexity of $O(m \cdot g(n))$.

In the following course we are going to discuss 3 methods to find a bounder for running time of $\sum_{i=1}^{m} t_i$ the running time of a sequence with m operations.

aggregate analysis. proof directly of a bounder on the running time of operations sequence. $\sum_{i=1}^{m} t_i \leq m \cdot g(n)$..

accouting method. in total the running time will be bounded by the ammount of money which we be given to the data structure. for operation i we get a_i *money, through this money we will pay for the total running time, the strctutr will hold bank which will update in each operation* $a_i - t_{i.}$. *if in the end of the running the bank is positive then, the total money* $\geq$ *total time running. i.e:*

$$\sum_{i=1}^{m} a_i - t_{i.} = \sum_{i=1}^{m} a_i - \sum_{i=1}^{m} t_i \geq 0$$

.

$$\rightarrow \sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} t_i$$

mthe potential method. we will define a potential on our data structure which is marked by ϕ. ϕ_i*is the potential after i operations.*

Definition. the amortized price a_i of each opertion is defined to be:

$$a_i = t_i + \phi_i - \phi_{i-1}$$

if $\phi_m \geq \phi_0$ then:

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} (t_i + \phi_i - \phi_{i-1})$$

$$= \sum_{i=1}^{m} t_i + \phi_m - \phi_0$$

hence,

$$\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} t_i$$

Exercise. propose a implementation to a stack which support the following:

$init(k)$ - intialize the stack with parameter k. $O(1)$.

time complexity: $O(1)$.

E.g if we do $init(4)$ then we can add only element $1, 2, 3$ i.e all till 4.

$push(x)$- add a element x to the head of the stack.

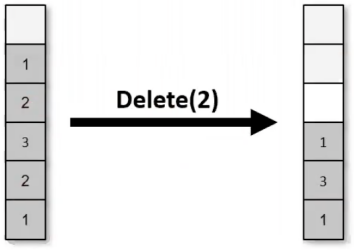
time complexity: $O(1)$ amortized with $pop()$.

$pop()$ - delete the element in the head of the stack.

time complexity: $O(1)$ amortized with $push()$.

$delte(x)$ - delete the elements from the stack with value x. After this call we can't insert to the stack element which have value x.

time complexity: $O(1)$.

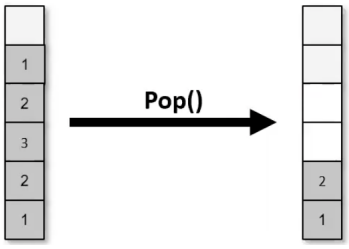


Solution. we want to show that for every sequence with length m of operations of $push, pop$, time complexity of each sequence is $O(m \cdot 1)$. hence, amortized time complexity of single operation is $O(1)$.

in order to solve the exercise, we will use a stack which S which will describe the stack. in addition, we will use a boolean array A in size of k, which will tell us for which x we call delete and $push(x)$ will be executed normal if $A[x] == false$.every time we want to execute pop for every element in the stack head which were deleted from the structure, and in the end we will delete additional element.

E.g look at the following example, we can see that a user init 5 to the stack and psu $1, 2, 3, 2, 1$ and he did delete $1, 2$ at some time, so we get like this:

x	A[x]
1	True
2	True
3	False
4	False



now we see that when we do pop we remove the element which was already deleted from the structure, in this case 1 is indeed deleted. so we deleted it and go to the next element which is 2 and it's also deleted then we remove it and continue to find element which is not deleted yet i.e 3 is not then we remove it and stop, hence get in the stack which give us the current situation $1, 2$.

Remark. in this implementation although the pop operation take $O(n)$ in the worst case since we can get a case in which all the element in the stack are deleted from the structure, we will see that the amortized complexity of the two operations $pop, push$ are $O(1)$.

- here we assume that every sequence of m operation start from emety structure.

aggregate analysis. we look at the following sequence which contain of m_1 psuh operation and m_2 pop operation s.t in the i call we removed k_i elements from the structure.

Example. look at : $m_1 = 4, m_2 = 3$.

$$\underset{O(1)}{push}\ \underset{O(1)}{push}\ \underset{O(1+k_1)}{pop}\ \underset{O(1)}{push}\ \underset{O(1)}{push}\ \underset{O(1+k_2)}{pop}\ \underset{O(1+k_3)}{pop}$$

good point. we can't remove element more than the number of elements which were inserted hence,

$$\sum_{i=1}^{m_2} k_i \leq m_1$$

in total we get:

$$\begin{aligned}\text{Total time } &= m_1 \cdot O(1) + \sum_{i=1}^{m_2} O(1+k_i) \\ &\leq O(m_1) + O(\overbrace{m_1 + m_2}^{m}) = O(m)\end{aligned}$$

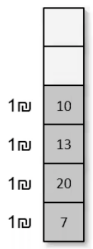
i.e although a part of the operation could be expensive, the complexity time which include a sequence of length m operations is $O(m)$.

accounting analyze. we will defined the following cost:

- $push(x)$ - $a_i = 2$ we will pay for insertion and pay in advance for future operations.
- $pop()$ - $a_i = 1$ we will pay only for cases in which the stack empty.

we need to show that after a sequence of $m$operations (which start from empty stack) our bank $\sum_{i=1}^{m} a_i - t_i$ will be in credit.

How we insure not getting into debt? for a $push(x)$ operation the cost in real is $t_i = 1$, and in every $push(x)$ we will pay 1 for insertion the the other shekel we will give as credit to the element x. now the $pop()$ operation which it cost in real is $t_i = k_i + 1$ will be paid using the k_i shekels which were given to k_i element as credit.



since,

$$0 \leq \sum_{i=1}^{m} a_i - t_i$$

we can bound the rumming time by:

$$\sum_{i=1}^{m} t_i \leq \sum_{i=1}^{m} a_i \leq 2m = O(m)$$

The potential method.

reminder. ϕ_i - the potential of the structure after i operations. The amortized cost a_i of the operation i :

$$a_i = t_i + \phi_i - \phi_{i-1}$$

if $\phi_m \geq \phi_0$ satsifed then:

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} t_i + \phi_m - \phi_0 \geq \sum_{i=1}^{m} t_i$$

How we are going to define our potential: Denote $\phi_i = |D_i|$

- $|D_i|$ - number of elements in the stack after i operations.
- we notice that $\phi_m \geq 0, \phi_0 = 0$.

we will show now that the amortized cost a_i is bouned by a constant for two operations.

the potantial method.

$$push(x) - a_i = t_i + \overbrace{(\phi_i - \phi_{i-1})}^{\text{number of elements inserted}} = 1 + 1 = 2$$

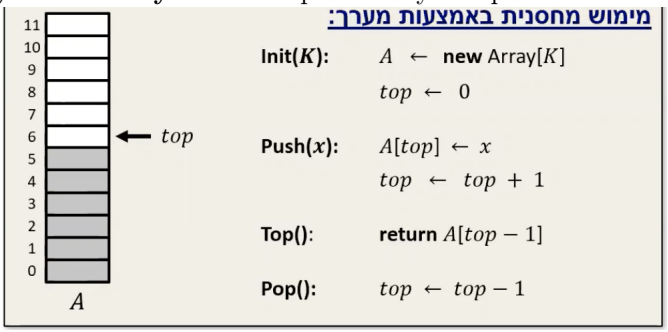
$$pop() - a_i = t_i + \overbrace{(\phi_i - \phi_{i-1})}^{\text{number of elements deleted}} = 1 + k_i - k_i = 1$$

Since $\phi_m \geq \phi_0$ (any time the stack will have 0 elements or more).

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i + \phi_0 - \phi_m \leq \sum_{i=1}^{m} a_i \leq_{explain} 2m$$

explain: since we in the worst case we have m insertion of elements.

Dynamic arrays. We saw previouesly a implementation for stack using a array.



Problem. this implementation limit us with intial size for the stack which is k. how can we change implementation to more dynamic s.t the space complexity will be $O(n)$?

Try 1. every time the array is filled, we allocate new array bigger by 1 *and copy the old array data to the new one. when the array empty, we will decrease it size in the same way, in order not to exceed space complexity of* $O(n)$..

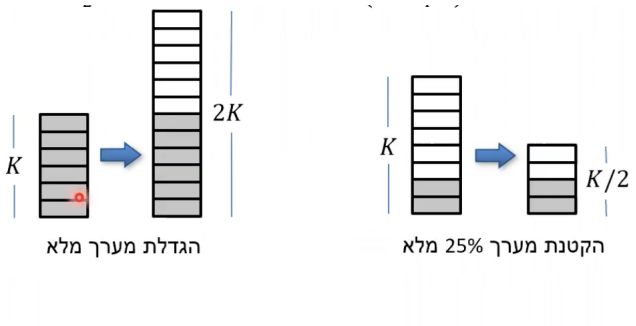
proplem in try 1. if the array were filled and we want to push n *elements and in each time we increase the size of the array by* 1*, the total time for* n *operations could be* $\Omega(n^2)$..

idea. we want to adapt the try 1 in way s.t the size of the array will be change rarely .

- I.e the number of $push, pop$ operations which are executed from the moment which a new array were allocated to new moment which we need to allocate again is big enough.

Rule of size change. denote by K *the aray size (we intialize* K *to be parameter).*

- When the array is filled $n = K$ we will alloacte new array with size $2K$.
- When the array is quartar filled $n = \frac{1}{4}K$ we allocate new array with size of $\frac{1}{2}K$.



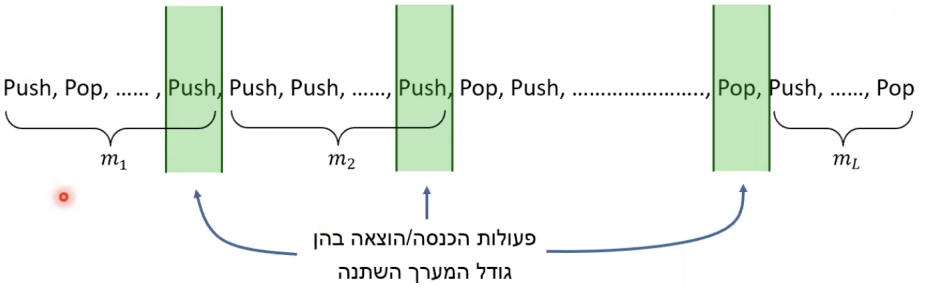
Remark. notice that the array will be always 50% filled.

Remark. assuming we have array with $\frac{K}{2}$ we need to execute $\lfloor \frac{k}{2} \rfloor$ push operations in order to get next increasing in size and $\frac{k}{4}$ operation of pop to get next size decreasing so in total in order to get a next change in size we need to execute $\Omega(k)$ operations of pop and push.

Worst case complexity analyze. Every operation of $pop, push$ will take at the worst case $\Theta(K) = \Theta(n)$ hence, a sequence of m operations of push and pop would take $O(m \cdot n)$ complexity.

aggregate analysis.

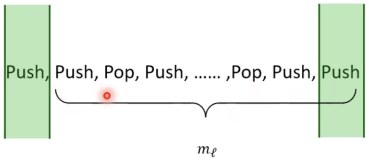
target: bound the running time of $\sum_{i=1}^{m} t_i$. We look at a sequence of m operations of push\pop as following:



when the push and pop operations (emphasized in green) illustrate the idea the a change in size were executed here and we hace a partitions of operation $m_1, .., m_l$ i.e we devide the sequence of m operations to L partiotions.

- each partition will contain a collection os push\pop operations from a changing size operation (not included) to the sequential changing size operation. I.e m_1 contain all the operation in the beginning untill a operation which change size, recall it $operation(1)$ and m_2 is all the operation executed after $operation(1)$ untill next operation which change size recall it $operation(2)$, and we keep doing that in assumption we have L partioitons.

Target. each partiotion in length of m_l take $O(m_l)$ time complexity. Then we conclude that a seqence of m opearion will run in $O(m)$ time complexity.



Denote by K the length of the table in this partiorion, Hence, the total running time in this partition is as following:

- m_l operations of$pop, push$ which take $O(1)$ time complexity each one.
- A changing size operation in the end of the partititon which take $O(K)$ time complexity.

Therefore, we deduce from above that: $m_l = O(K)$ i.e $K = O(m_l)$.

- we need to execute at leat $\frac{K}{2}$ insertion operation or $\frac{K}{4}$ pop operatons between two sequential changing size operation. otherwise we get no size change therefore, $m_l = 0(K)$

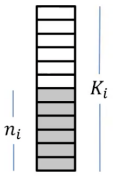
Hence, the total running time is:

$$O(1) \cdot m_l + \overbrace{O(K)}^{\text{Changing size operation}} = O(m_l)$$

The potential method.

Intiuition.

The potential of the structure is the bigger when we change in size of the dynamic array. Moreover, the potential is lowest after we change in size since there is a lot of operation to do till next size change.



denote by n_i the number of operation after i operations and k_i is the size of the array at this stage. Define the following potential function:

$$\phi_i = |n_i - \frac{k_i}{2}| \cdot c$$

when c is constant will be determined in advance. We will show that for every opertaion in the structure satisfied:

$$a_i \leq 1 + c$$

Remark. here as in previos question our potential is the number of elements after i operations. but here changing size is little bit problem to define our potential.

The amrtized prices of insertion\deleting with no size change.

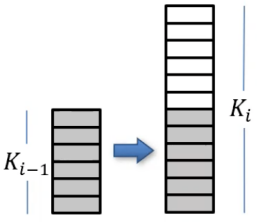
$$t_i = 1$$

$$\phi_i - \phi_{i-1} \leq c$$
$$a_i = t_i + \phi_i - \phi_{i-1} \leq 1 + c$$

notice that it's true for every c.

The amrtized prices of insertion\deleting with increasing size change.

- Before opertaion: $n_{i-1} = k_{i-1} - 1$.
- After opertion: $n_i = \frac{k_i}{2}$.



m

hence, satisfied that:

$$t_i = k_{i-1}$$

$$\phi_{i-1} = |(k_{i-1} - 1) - \frac{k_{i-1}}{1}| \cdot c = (\frac{k_{i-1}}{2} - 1) \cdot c$$

$$\phi_i = |\frac{k_i}{2} - \frac{k_i}{2}| \cdot c = 0 \text{ (equivalnce point is } \frac{k_i}{2})$$

Hence,
m

$$a_i = t_i + \phi_i - \phi_{i-1} = k_{i-1} - (\frac{k_{i-1}}{2} - 1) \cdot c$$

$\forall c \geq 2$ we get $a_i \leq 1 + c$.
E.g take $c = 2$ therefore we get:

$$a_i = -2 \leq 1 + 2 = 3$$

The amrtized prices of insertion\|deleting with decreasing size change.

- Before opertaion: $n_{i-1} = \frac{k_{i-1}}{4} + 1$.
- After opertion: $n_i = \frac{k_i}{2}$.

$$t_i = k_{i-1}$$

$$\phi_{mi-1} = |(\frac{k_{i-1}}{4} + 1) - \frac{k_{i-1}}{2}| \cdot c = (\frac{k_{i-1}}{4} - 1) \cdot c$$

$$\phi_i = |\frac{k_i}{2} - \frac{k_i}{2}| \cdot c = 0 \text{ (equivalnce point is } \frac{k_i}{2})$$

$$a_i = t_i + \phi_i - \phi_{i-1} = k_{i-1} - (\frac{k_{i-1}}{4} - 1) \cdot c$$

$\forall c \geq 4$ we get $a_i \leq 1 + c$.
E.g take $c = 4$ therefore we get:

$$a_i = -4 \leq 1 + 4 = 4$$

General time analyze.

- for every operation i we got $a_i \leq 1 + c$ (for choosing $c \geq 4$ and it cover all the cases above).
- the first potential ϕ_0 is bounded by constant , sice the first table size is constant.
- $\phi_m \geq 0$ by the definition of the potential. (number of element in the stack is at least 0).

Hence,

$$\begin{aligned}(c+1) \cdot m \geq_{\text{proved above}} \sum_{i=1}^{m} a_i &= \sum_{i=1}^{m} t_i + \phi_m - \phi_0 \\ &\geq \sum_{i=1}^{m} t_i - \phi_0 \\ \Rightarrow \sum_{i=1}^{m} t_i &\leq (c+1) \cdot m + \phi_0 = O(m)\end{aligned}$$

I.e the running time of sequence with m operation is $O(m)$. Namely. the amortized time for operation is $O(1)$.

Hash Tables.

What is the best way to implement dictionary.

Implement using arrays.

- Or key is the index.
- The information which the array hold is the data.
- what is the problem?
- it requires keys in domain of $0, , , m - 1$.

Problems.

- Sometimes the size of the domain of the keys values is way bigger than number of keys in which we use, which let us waste memory.

  Example. number of ID's which are from 9 digits. I.e 10^9 keys but in israel we have less than 10^7 people. Hence, the uses of the array will explit less than 1% of the memory allocated.

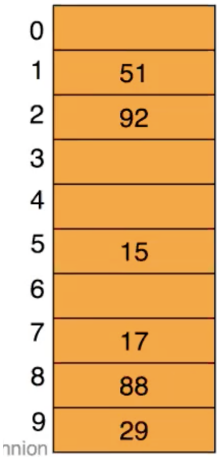
  Example. Number of strings of hebrew letters with length of 30, Through it we can describe a private name, middle name, and last name of israeli citizens. is more than 22^{30} when numper of people is less than 10^7.

Remark. The three operations is eecuted in time of $\Theta(logn)$ when we use AVL tree, or skip lists, so why we use other structures?

Hashing. Implementing a dictionary using a direc accessing: the kwy itself is used as a index in the array. When the space of the keys is big we will calcuate then index using hashing a function. Our target is to implement searching, inserting, deleting operation is average time of $O(1)$. We will define a hash function $h : U \to \{0, ..., m-1\}$ when given a key in domain U it calculate the index in the compatible domain. I.e the index of key k will be $h(k)$.

Example. look at $m = 10, h(k) = kmod10$.

Insertion input is $51, 17, 15, 92, 88, 29$.

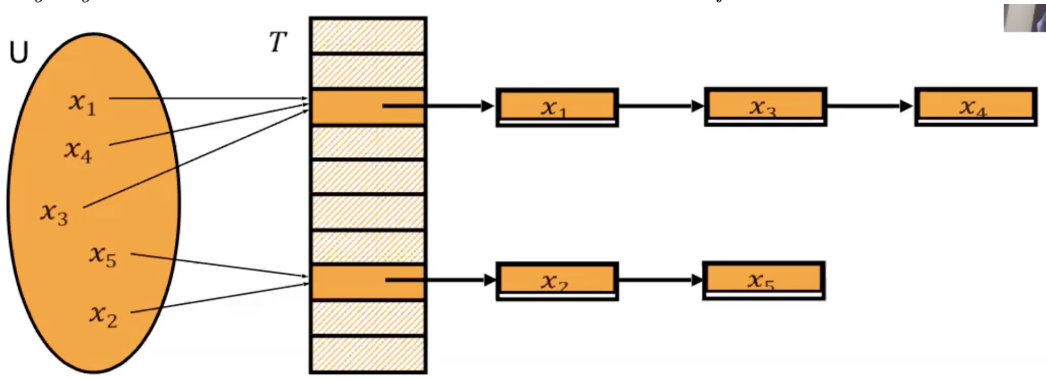


we can see that $h(51) = 51mod10 = 1$ therefore, we put in the index number 1.

Problem. In the hash problem there could be crash then $x \neq y$ but $h(x) \neq h(y)$. E.g we can look at $h(81) = 1 = h(51)$.

- How are we going to solve this?
- How are we going to choose hash function?
- Universal hash: using a coin rolloing in order to improve the hash properties.

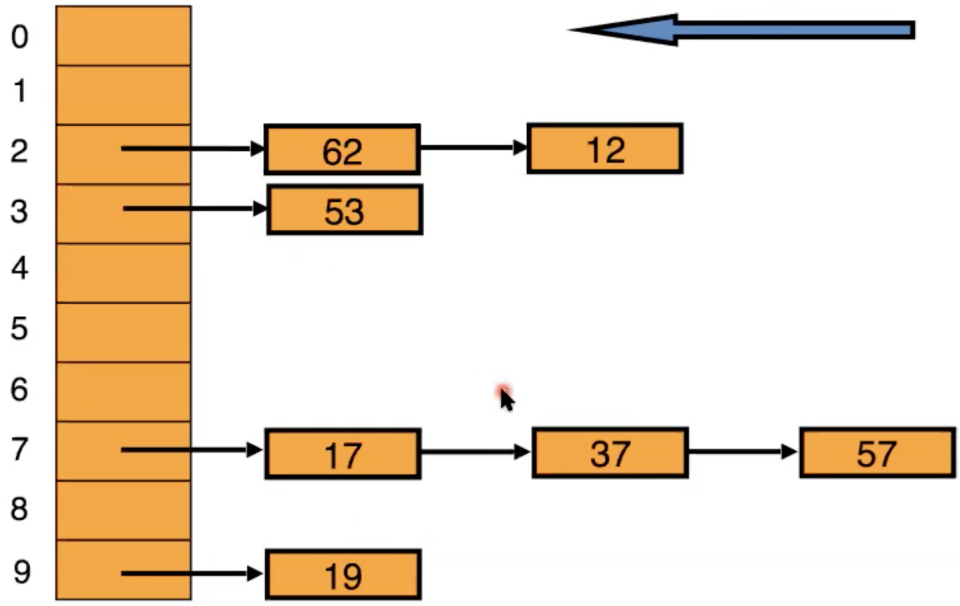
Solution. every keys which have the same index we will build a link list of them as following.



Operation implementation.

- $Insert(T, x)$: insert x to the head of the list $T[h(x.key)]$. The time complexity is Θ(list length)in the worst case.
- $Search(T, x)$: Search the elemnt with the key k in the list $T(h(k)$. The time complexity is Θ(list length) in the worst case.
- $Delete(T, x)$: Delete x from the list $T(h(x.key)]$.The time complexity Θ(list length) in the worst case.

Example. Assuming $m = 10, h(k) = kmodm$. Input $53, 62, 17, 19, 37, 12, 57$.



Remark. $m = 10$ we will choose for the question requirements. We will see later more good choice $m = 11$.

- It's efficient to calculate.
- Scatter well.

  - Best away: a totally random function from the domainn U to our range $\{0, 1, ..., m-1\}$.
  - Meaning: every element in the domain is mapped to random element in the range with no independent to the other elements.
  - Problem: this is a big family of funciton and there is no way to describe it efficiency of a function in a set. we will try to get close to function like that later.
- We will start with very comfort assumption of scatter: The assumption of the uniform simple scatter:
  - Assuming that a key choosen randomly form U with independtly to the other keys.
  - Assuming that h map a random key to random cell independently in other keys.

Remark. We could put trees instead of linked lists.

Complexity. In the worst case at most all the elements are inserted to the same list and the operation time $\Theta(n)$. The advantage in hash is the average time for operation. In order t doit we will choose a hash function which scatter well the keys to different lists.

Definition. We say that hash function satisfy the assumption simple uniform hash of h map random element from the domain in uniform probability $\frac{1}{m}$ for each in the matching range and independently to the other elements.

We will analyze the searching time under the assumption of uniform simple scatter.

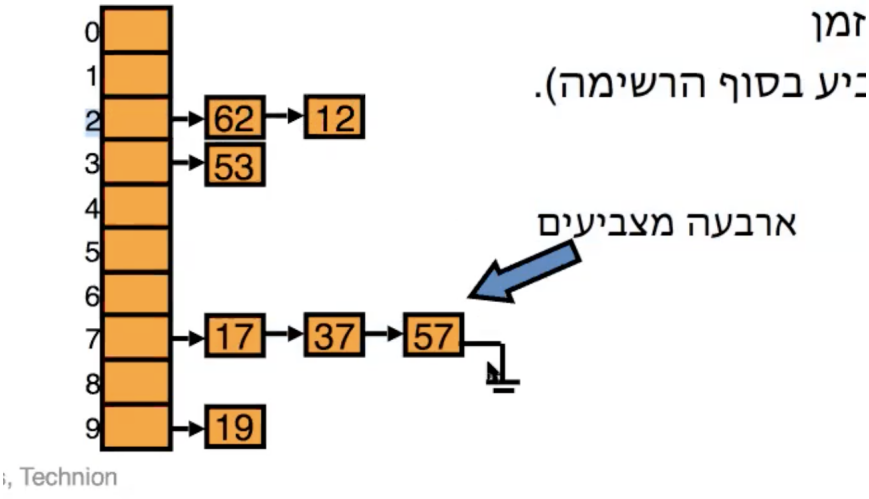
Definition. let n be the number of the keys in use and m is the table size. The load factor is defined by $\alpha = \frac{n}{m}$.

Remark. Under the assumption of simple uniform scatter the average length of a chain is α. Since the elements are devided equally between the different chains.

Theorem. *In the method of chains and under the simple scatter assumption, "time" of a failed searching is $1+\alpha$. (I.e searching in which we are searching for key which doesn't exist in the table).*

Proof. Notice that: □

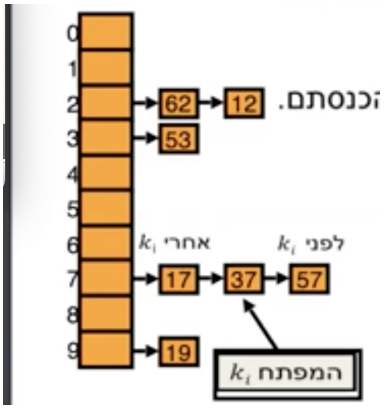
- In the assumption of simple uniform scatter every key reach randomly to one of the m lists.
- The time take for fail searching is in average searching at ont of the least till it end.
- The average length of list in assumption of uniform scatter is $\alpha = \frac{n}{m}$.
- Hence, in average it require $(1+\alpha)$ time (Also include the time of checking the pointer in the end of the list).



Theorem. *In the chains method and under a simple uniform scactter, the expectatiom of successed searching is less than* $1 + \frac{\alpha}{2}$. *(I,e in average we going to find this element in the middle of the least).*

Proof. (Theorem above). □

- let $k_1, .., k_n$are the keyss in the table while searching time. by there insertion order.
- What is the average searching time of key k_i.
- After this key which we are looking for $n - i$ keys are inserted.
- Hence, in averge the size of the left list for the key k_i is $\frac{n-i}{m}$.
- From here the average seaching time of a key k_i is $\frac{n-i}{m} + 1$.



The average searching time t for key will be as following (it's for all the keys no matter where they exists in the list).

$$t = \frac{1}{n}\sum_{i=1}^{n}(1 + \frac{n-i}{m}) = 1 + \frac{1}{n \cdot m}\sum_{i=1}^{n}(n-i)$$

$$= 1 + \frac{1}{n \cdot m}\frac{(n-1)n}{2} = 1 + \frac{\alpha}{2} - \frac{1}{2m} \leq 1 + \frac{\alpha}{2}$$

Remark. The lists are not ordered in specific order.

Corollary. *When the order of numer of keys n which we use is m as the size of the array i.e $n = O(m)$ we will get that the load constanr is constant i.e $\alpha = O(1)$. Hence, in the cahin method, uner the assumption of a uniform simple scatter. all the operation take $O(1)$ in average.*

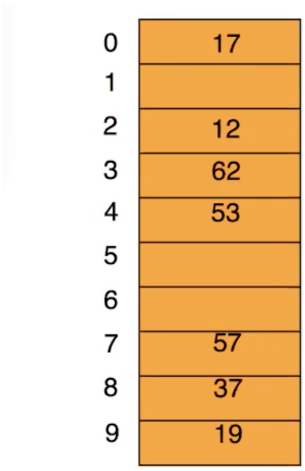
Example. For 2100 keys from a domain U of natural numbers, we will say 10^6 we can hold a array in which 700 places and in averge the length of every chain is 3 and identical for the searching time.

Solution for crash with no chaing (Open Addressing). In the following method we are not going to use chains. Instead, all the elements will be inserted to the table. It's obvious that the load factor is less\equeal than 1 ($n \leq m$). Since we are not going to solve the problem with lists, we propose a different solution for crash.

(1) Linear Probing. (scan).

Linear Probing. In linear probing, while inserting, if the wanted place $h(k)$ is used, we will try the next place $mod m$ and the continue..

Example. $h(k) = kmodm, m = 10$. Input: $57, 12, 37, 19, 17, 62, 53$.



E.g in the follwing input first all the array is empty, we insert 57 we see that $57 mod 10$ is not used so we insert it to the index 7 now we go with 12 notice that $12 mod 10 = 2$ is also not used therefore, we put it there. now we got 37 notice that $37 mod 10 = 7$ but the index 7 in the array is already used. Hence, check if it's next index not used i..e 8 and it's indeed not so we insert it there. after that we insert 19we see that $19 mod 10 = 9$ is empty therefore, we insert 19. Afterward, we see that we insert 17 but $17 mod 10 = 7$ is used and the next index is used and the next index which is 9 is also used and it's our final index, in this case we search cyclic i.e we got to the first index which is 0 and doing the same we notice that 0 index has no data so we insert to it 17 .Moreover, seaching is in the same method as inserting.

Deleting in open addressing method. What if for example a element were used E.g we had 57 in the index 7 place then we insert 37 but it's used. Hence, check if it's next index not used i..e 8 and it's indeed not so we insert it there, after that we

insert 19 and then 17 as previous but now we try to execute the following first, we are going to delete 37 and search for 17. if we delete 37 then we see that $37 mo 10 = 7$ will be empty and when we are going to search for 17 we see that $17 mod 10$ is empty so we return that 17 key is not found, when indeed it were inserted in index 0. So to solve this problem in each place we delete we are going to jold data which tell us if in the specefic index a element were deleted in order to keep searching in next indexes.

So in general how we delete element? We can't delete a element in which the searching chain we be disconnected from other elements.

- Solution: Denote the place in which element were deleted by *delete*.
- In time of searching of x in case we see *delete*,we will continue seaching untill finding x or a index which is marked by NULL (i.e index which no element were deleted in it's place).
- While inserting: in case we got mark *delete*, we will use it's place to put x.

Pros and cons of the linear probing.

- The first advantage of linear probing is simple.
- Linear probing tend to fill blocks since when a block exist, the next elements are going to join it in higher probability than filling a new place. Hence, linear probing constitute a good solution for the uniform scatter assumption.
- When our use require a deleting, the length of searching depend also on the elements were delete and not onlt the one who exist currently in the structure.

Example of using dictionary with no deleting.

- Table of variables names in running code.
- ID's numbers which are not used again. (i.e someone take dead person ID so it's used again since the first person id should be deleted and given to other person).

We will describe now other methods for open addressing which explit better the uniform scatter assumption. It's better to use those methods when there is no deleting. When there is need in deleting, the linked list method work better.

Rehashing. In this method we are going to define more the one hash function and if a index were used and we try to insert other element to it, we use other hash function and if it still used and we use again other hash function i.e we use sequence of infinite hash function $h_0, h_1, h_2, ...$ we will try to hold x iin the place $h_0(x)$ if used then we try in the place $h_1(x)$ we will conitnue till we success.

Example. In the linear probing method $h_i(x) = h(x) + i$.

What is insertiong average time.

- Under the assumption of uniform scatter the probability that a place is used is α.
- The probability that in the i first trials that the place is used is α^i (with assumption that the hash function are independent).
- Notice that it's exactly a random variable $X \sim Geo(\alpha)$ since we are searching for the first trial which we have succeess from infinite trials, and this problem is equivalent to a random variable distribute geometric.

- Hence, the probability of searching exactly i places is $\alpha^{i-1} \cdot (1-\alpha)$.
- From here the length of searching under the assumption of a uniform scatter is:
$$l = \sum_{i=1}^{\infty} i \cdot \alpha^{i-1}(1-\alpha) =_{\text{expectation of geomtric r.v}} \frac{1}{1-\alpha}$$
- E.g for $\alpha = \frac{2}{3}$ the length of searching average is 3 i.e we walk through 3 places till we find empty index.

Remark. Also in this implementation, the deleting is done by marking in *delete* after we delete.

Double hashing. We can get the same result of rehashing by only using 2 hash function d, h and not sequence of hash function. when

$$h_i(x) = h(x) + i \cdot d(x)$$

i.e each time we insert we jump by $d(x)$ times i when every try i we try to insert.

Remark. h, d are chose in independent way.

What is the relation between $d(x)$ to the table size m?. The size of the table and $d(x)$ should be disjoint number s.t $h_0(x), .., h_{m-1}(x)$ will cover all possible indexes in the domain $\{0, 1, ..., m-1\}$. Hence, it's easier to choose m as prime number.

Remark. it's easier since all the primes number are odd expaect 2. So we can reach m every time we jump. E.g look at $m = 8.d(x) = 2, h(x) = 1$ we get the following:

$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
1	3	5	7	1

now for $i \geq 4$ we will get cyclic values and we need to execute for $i = 5$ in order to see if we got index which we already pass on in this case we see 1 when $i = 5$ as the first cyclic value so we stop

$i=5$	$i=6$	$i=7$	$i=7$	$i=8$
1	3	5	7	1

what happen if we choose m prime assuming $m = 7$ we get:

$i=0$	$i=1$	$i=2$	$i=3$
1	3	5	7

In this case we didn't need to check cyclic values since we chose m as prime so it save time for searching.

Remark. Also in this implementation, the deleting is done by marking in *delete* after we delete.

Hash Functions.

requirements from hash function.

- Scatter well.
- Easy to calculuate.

Deviding method modulo m..

$$h(x) = x(modm)$$

For the following hash function, preffered that m :

- Power of 2 or 10. Since in power of 2 the hash function rely only on the $log_2(m)$ first bits LSB. Moreover, in power 10, the hash function rely in the first $log_{10}(m)$ first digits. It's more wanted that our hash funcction use all the informarion in the key in order to make us more close to the uniform scatter assumption. Now this is rational since in average the most of the number are power of 2 then the most of the numbe rin our input would be inserted to to the same cell number 0 and it's not helping for scatter, namely, when our hash function is $h(x) = xmodm$.
- Prime (we say why it's important previously) which is not power of 2. A power which are close to 2 leads to non uniform scatter when the keys are written in a power of 2. E.g strings are written in base $2^8 = 256$.

Remark. Less terrible, because you will think that in our lives numbers that are strong of 10 are very common (large sums, for example, there are more products at NIS 100 than at NIS 97, etc., etc.). In addition, if the hash table deals with things related to the computer, then the powers of 2 are very common (because everything is binary).This is not the same situation with other powers.

The multiplicity method for constant $0 < a < 1$..

(1) Multiply the key k by constant a.
(2) Find the fraction part of the result.
(3) Multiply the fraction part by m and round it down:

$$h(k) = \lfloor m \cdot frac(a \cdot k) \rfloor$$

The value of m is not critical. Good value of a which leads good scatter is:

$$a = \frac{(\sqrt{5}-1)}{2} = 0.61803 \text{ the triangle relation (fibonacci)}$$

Example. $m = 1000, a = 0.61803, k = 123456$.

$$h(k) = \lfloor m \cdot frac(123456 \cdot 0.61803) \rfloor$$

$$= \lfloor 1000 \cdot 0.0041151 \rfloor = \lfloor 41.151 \rfloor = 41$$

Remark. This is good scatter. I.e for different number high probability they will be mapped to different places in our tabel.

What if out input is strings?

Consdier string as a number in ascii code i.e:

$$'a' = 97 = 0110\ 0001$$

$$'b' = 98 = 0110\ 0010$$

Naive solution: executing xor bit by bit. E.g

$$h(\text{"}ab\text{"}) = h((0110\ 0001)xor(01100010)) = (0000\ 0011) = 3$$

. Disadvantage:

$$h(\text{"}aa\text{"}) = h((0110\ 0001)xor(0110\ 0001)) = 0000\ 0000 = 0$$

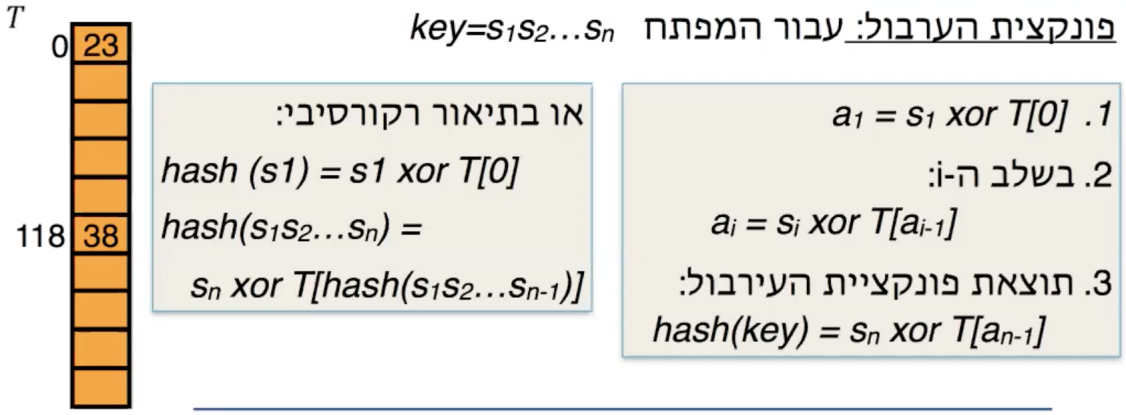
$$h(\text{"}bb\text{"}) = h((0110\ 0010)xor(0110\ 0010) = 0000\ 0000\ = 0$$

(1) I.e get the result 0 if two letters appear even times in the string. For example $h(\text{"}abccba\text{"}) = 0$. Hence, even though *xor* seem to us the it give us a uniform scatter and good, we got that a family of strings now mapped in a unique way. Notice that our probelm weith strings like $aabb, baba, cddcakka$ i.e string which every letter in it appear twice. In particular we can look at the set of words

$$w_n = \{a_n | a_n \text{ is alphabet}, \wedge | a_i \neq a_j \in a_n, \forall i \neq j | \text{ is not even}\}$$

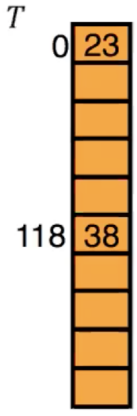
(2) The range of values is limited: $h(x) \leq 255 = 2^8 - 1$.

Preffered method. Define a random permuatation $(\pi_0, ..., \pi_{255})$ of $0, 255$ and store it in a array. Our hash function for a $key = s_1 s_2 ... s_n$.



In this method we rely on previous hash or in general our hash function defined recursively.

Example. Here we are going to rid of the problem which we had earlier when every letter appear in the string even time we will see how.



Now first we took a random permutation from number in our range to the palace in array.

$$\pi : \{0, ..., 255\} \to \{T([0], ..., T[255]\}$$

and put it in pur array as we see $\pi_0 = 23, \pi_{118} = 38$.

We want to calculate $h(aa)$:

$$hash(a) = T[0]xor97 = 0001\ 0111 xor\ 0110\ 0001\ = 0111\ 0110 = 118$$

Now we are not going to get 0 for $h(aa)$ since we have by definition of our hash function:
$hash(aa) = T[hash(a)]xor97 = T[118]xor97$

$$= 0010\ 0110\ xor\ 0110\ 0001 = 71$$

Remark. For other permuation we got other hash function.

Our next problem (small range).

- In general if we have hash functions $h_1, .., h_k$ in which each one has small range and they are independent then we can connect them.
- I.e $h(k) = h_1(key)||h_2(key)||...||h_k(key)$.

In order to solve the problem of range we can use two permutations T_1, T_2 and thread the result.

$$hash(k) = hash_1(k)||hash_2(k) = 256 \cdot hash_1(k) + hash_2(k)$$

The size of our new range is the size of the hash table, $256^2 = 2^{16}$ (When sice of each T_i os 256). We can get the want range by using small number of other permutations.

Remark. Same result we get if instead of using $hash_2$ we add 1 to the first letter of the string and use again the same $hash_1$ function

Example. $hash(acb) = hash_1(acb) \cdot 256 + hash_1(\underbrace{\ b\ }_{a+1} cb)$.

- In general we want that load factor to be constant. The size of the table $O(m) = n$ then $\alpha = \frac{n}{m} = O(1)$.

Problem. It require to know in advance the number of elements n which are going to be inserted to the structure.

Solution. When the hash table "full" we do expensive operation but rare.

- Allocating new table in a double size.
- Inserting all the element to the new table.
- We give up about the old table.

Cost: even though the following operation is expennsive, but the amortized time of the operation (including increasing the table) is $O(1)$..

Universal Hash functions.

- For every choose of hash function, exists bad sequence of keys in which all of them we be mapped to the same place.
- Tell now we said that our analyze we work if we choose key in U randomly and in independent way to other keys, and our hash function will scatter well.
- In general a key won't be chosen randomly, and we can't insure independence.

- E.g there is probability that our user code in sequential way our code variables and unfortunately our hash functionw e build the symbpol table map all this name to the same place in the hash table. Hence, working with every program of this user won't be effecient.
- We want more stable module.

Choosing hash function randomly.

- Given hash functions we will choose hash function randomly from this set off functions.
- Then we can say that our system behaive well for every sequence of keys. In case the choosing of hash funciton won't rely on keys. Since, the user can't know which hash function are we using by choosing it randomly without letting him now and try to make our system fail.
- We can look at the following:
  - Someone choose keys, then we choose randomly hash function in the set, we map the keys and check if crash occured.

Definition. let H be set of hash function form domain U to the set $\{0,..,m-1\}$ the set H is called universal if for each different pairs $x,y \in U$ number of function H in which $h(x)=h(y)$ is $\frac{|H|}{m}$.

Distinction. The probability p that in random choice of hash function from H, a key x will crash with other key y is:

$$p = \frac{(\frac{|H|}{m})}{|H|} = \frac{1}{m}.$$

we will show that a use in a universal set lead to good scatter. Afterward, we will build this set.

Intiuition.

- In assumption that our uniform scatter if the input disribute in set U then:

$$\forall i \in \{0,..,m-1\}, prop[h(x)=i] = \frac{1}{m}$$

- And for universal hash function:

$$\forall x,y \in U, x \neq y, Prop[h(x)=h(y)] = \frac{1}{m}$$

Theorem. *let H be universal set of hash function to table T in size m. If h we will choose random;y from H and we will use ot in order to hash n keys, then for every key, the expected number of crashes in the method of the linked lists is equal to $\frac{n-1}{n}$.*

Proof. We saw that the probability of crash of key x with other key y is $p = \frac{1}{m}$. The expectaiton of number of crashes of specific key x with other keys is:

$$\bar{L} = \sum_{y \in T, y \neq x} Prop[h(x)=h(y)] \cdot 1$$

$$\bar{L} = \sum_{y \in T, y \neq x} \frac{1}{m} = \frac{n-1}{m} = \alpha - \frac{1}{m}$$

□

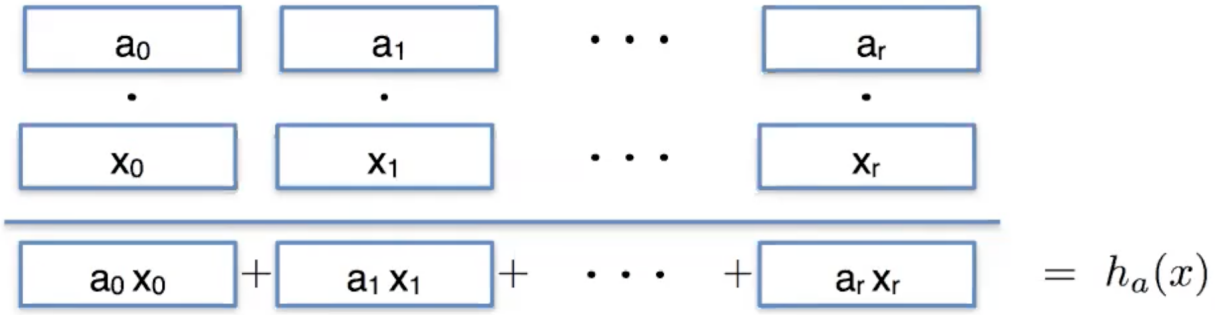
Corollary. *Number of crashes expectied to each key is less than our load factor.*

Constructing a univeral set.

$$h_\alpha(x) = (\sum_{i=1}^{r} a_i \cdot x_i) mod m$$

- The functions we present is a linear combination of the key parts.
- We devide the key x ro $r+1$ parts and calculaute the combination.
- The function discription is $a_i : a = (a_0, a_1, ..., a_r)$.

Remark. Every change with the value of a we get other function.



- We will choose the size of the table to be prime number m.
- We will breal every key x to $r+1$ parts in a constant length $x = [x_0, .., x_r]$ (E.g in length of 8 bits), S.T $x_i < m$ when m is the table length.
- For every sequence of $a = [a_0, .., a_r]$ from the domain $\{0, ..., m-1\}^{r+1}$ we will define hash function as following.

$$h_\alpha(x) = (\sum_{i=1}^{r} a_i \cdot x_i) mod m$$

- The set of function H is $\bigcup_a \{h_a\}$ and number the function in it is m^{r+1}. Since we have m values for a_0 and other m for a_1 till $r+1$ we get $\underbrace{m \cdots m}_{r+1 \text{ times}} = m^{r+1}$.

Example. Assuming that the function $a = [248, 223, 101]$ were chosen, we determine $m = 5 - 3$ and the range of keys $\{0, 1, ..., 2^{24} - 1\}$ in order to caclulate the function on a key with 24 bit. We will break the key into 3 parts of a 8 bits. E.g given a key $x = 1025 = [0, 4, 1]$ we will cacluclte that hash $h_a(x)$ by the formula

$$248 \cdot 0 + 223 \cdot 4 + 101 \cdot 1) mod 503 = 490$$

notice that x is calculated as following since we using 24 bit we can represent as three parts of 8 bits therefore we have $2^8 - 1 = 255$ so for $x = 1025 = 1 \cdot 256^0 + 4 \cdot 256 + 0 \cdot 256^2$

Remark. Using this family: when our user define hash table T in size of m, the program grill number a and use a hash function h_a for every operation executed in the following hash table.

Theorem. *The set of funtions $H = \{h_a\}$ is a universal set.*

Proof. let $x = [x_0, ..., x_r]$ and $y = [y_0, ..., y_r]$ different keys. WLOG we assume $x_0 \neq y_0$ explicitly. $x_0 \neq y_0 modm$. (since $x_0, y_0 < m$).

- We need to show that the number of functions h in set in which $h(x) = h(y)$ is $\frac{|H|}{m}$.
- Every function h_a is determined in set H by choosing $r+1$ values: $a_0, ..., a_r$ in the domain $\{0, .., m-1\}$.
- Hence, number of functions in H is m^{r+1}. And we need to show that m^r satisfy $h(x) = h(y)$.
- We will see that every choose for $a_1, .., a_r$in domain $\{0, ...m-1\}$ there is exactly one a_0 in the domain in which $h(x) = h(y)$. Since if that a_0 satisfy this for choosing $a_2, ...a_n$and each has m options then a_0 satisfy it to m^r function.
- I.e it's enough since then we get that the function $h_a$which sartisify $h(x) = h(y)$ is m^r as required.

Lemma. *for every choosing of $a_1, .., a_r$ in the domain $\{0, .., m-1\}$ exists a_0 unique in the domain in which $h_a(x) = h_a(y)$.*

Proof. (Lema).

$$h_a(x) = h_a(y) \iff \sum_{i=0}^{r} a_i \cdot x_i (modm) \equiv \sum_{i=0}^{r} a_i \cdot y_i (modm)$$

$$a_0(x_0 - y_0) \equiv -\sum_{i=0}^{r} a_i(x_i - y_i)(modm)$$

Since m is prime, then for every number which is not 0 mod m we get inverse, hence:

$$a_0 \equiv [-\sum_{i=0}^{r} a_i(x_i - y_i](x_0 - y_0)^{-1}(modm)$$

□

Remark. The conidtion that $x_0 \neq y$ (WLOG) is that we can insure that $x_0 - y_0$ has inverse in $modm$ since 0 has no inverse in any field.

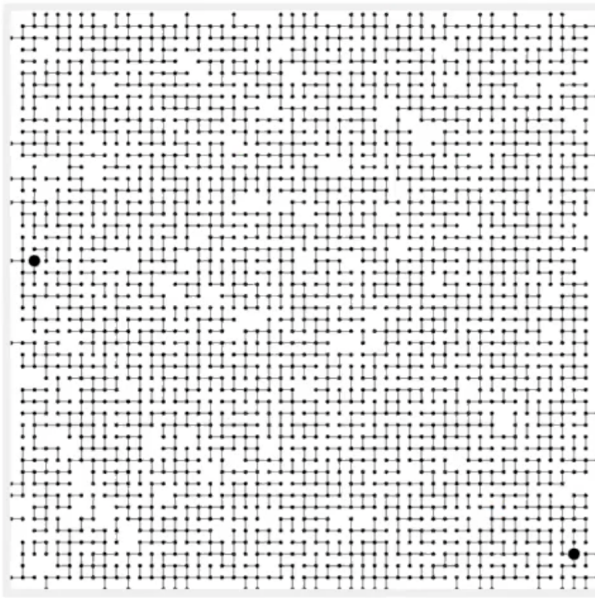
- We defined a universal set of hash functions.
- We show that for random functions in the set the expectations of crashes number is less than the load factor $\frac{n}{m}$.
- We build specific set of functions:

$$h_a(x) = \sum_{i=0}^{r} a_i \cdot x_i (modm)$$

- We also show that the number of function which satisfy the definition, for a choosen randomly in $\{0, .., m-1\}^{r+1}$.
- Remember our target was avoiding the user to know what our hash function in which he can put input to ruin it's effecient in this case we choose random hash functions in which the user could never predict which is pretty good.

□

Union find.



Riddle. Is there a path from the left point to the right point?

Disjoint sets - Union/Find problem. Given a world of elements U, $|U| = n$, For comfort reasons we assume that $U = \{1, 2..., n\}$. Data strcutre which support disjoint set support the following operations.

- $Makese(i)$ - return a new set which has single element i.
- $Find(i)$ - return the set in which i beloong to.
- $Union(p, q)$ - Union the the sets p, q. I.e return new set which has the union of element in the set p, q. (The source sets p, q are ceddatiaed from existance).

Example. Assuming we have noew the following sets $\{1, 3\}, \{5\}, \{2, 4, 6, 7\}$.Then we do the following operations.

(1) $p = find(6)$.
(2) $q = find(5)$.
(3) $r = union(p, q)$.
(4) $s = find(6)$.

Result. after the following operation we see that r will have $\{5, 4, 4, 6, 7\}$. The variables p, q will not hold more sets. The variable s we hold the same set as r..

Example. Given n cities $\{1, 2, ..., n\}$. In the beggining all the cirties are disconnected. Every period they build a direct street between twp cities. There is need to support the following operation.

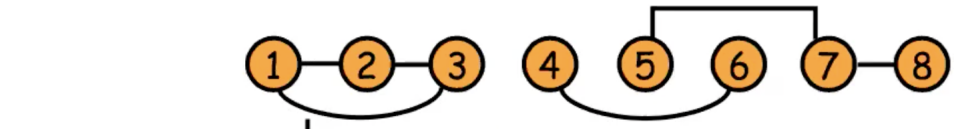
- $Add - Road(x, y)$ - add street between city x and y.
- $Check-Connectivity(x, y)$ - determine wether two cities x, y are connected in such a path.

Solution. Intializing: we will make n sets $\{1\}, .., \{n\}$ by the following loop:

for(i=n;i≤n;i++)

$Makeset(i)$

Now $Add-Road(x,y)$ is implemented by $Union(Find(x), Find(y))$. when $Check-Connectivity(x,y)$ if $Find(x) = Find(y)$ return true.



הכביש שהוסף	אוסף הקבוצות הזרות							
	{1}	{2}	{3}	{4}	{5}	{6}	{7}	{8}
(1,2)	{1,2}		{3}	{4}	{5}	{6}	{7}	{8}
(1,3)	{1,2,3}			{4}	{5}	{6}	{7}	{8}
(2,3)	{1,2,3}			{4}	{5}	{6}	{7}	{8}
(5,7)	{1,2,3}			{4}	{5,7}	{6}		{8}
(4,6)	{1,2,3}			{4,6}	{5,7}			{8}
(7,8)	{1,2,3}			{4,6}	{5,7,8}			

, Technion

First Naive implementation, We will use a array from type SET in size of n and in the structure there is *counter* intialized to 0. In the place $A[i]$ we will stote the set name which the elemnt i belong to. In the following implementation SET is simply int. E.g $\{1,3\},\{5\},\{2,4,6,7\}$ is represent by the following array.

	1	2	3	4	5	6	7
A	4	12	4	12	5	12	12

All the element in the same set got the same set name which is number in this case. E.g $1,3$ gor 4 since they are in the same set.

(1) $Makeset[i]$ is implemented by $A[i] = ++counter$.
(2) $Find(i)$ is by $A[i]$.
(3) $Union(p,q)$ is implemented by incrementing the counter by 1, and walk on the array and write the value of the counter in every place in which p,q is written, the funtion return the value of counter.

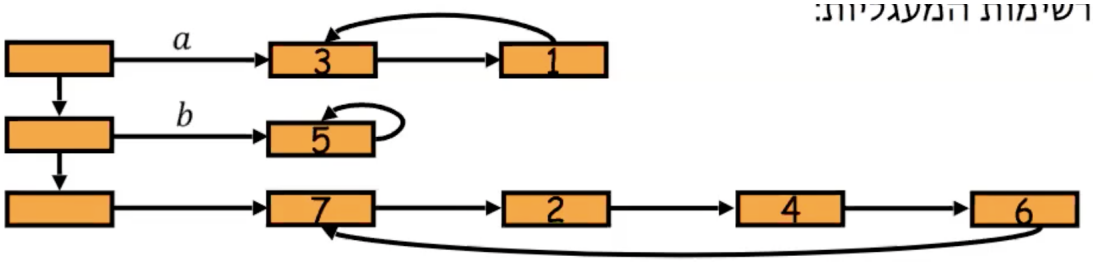
Example. After $Union(p,q)$ with $q = 5, p = 4, counter = 12$, after union the following sets we get:

	1	2	3	4	5	6	7
A	13	12	13	12	13	12	12

Notice that all the element which were in set 4 and in set 5 will be in one set which is 13 now.

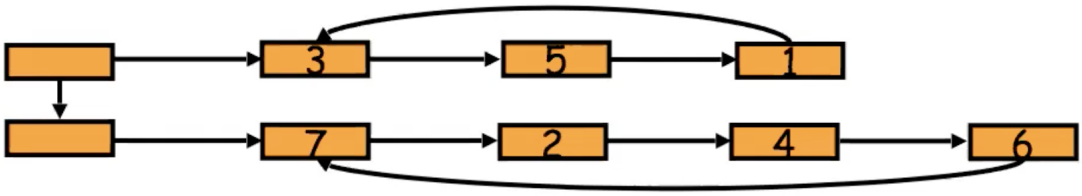
Time complexity: Find and Makeset require $O(1)$ time and Union require $o(n)$ time.

Other naive implementation. We will represent each set as a cyclic list. We will hold a linked list of pointer to the cyclic list.



$Union(p, q)$ - is implemented by union of the lists which point by p, q. Time $O(1)$. In this use SET is a pointer to node with type NODE.

Example. After $Union(a, b)$ we get:



- $Find(i)$ - Is implemented by walk on all the lists till we find the element i. Time $O(n)$.
- $Makeset(i)$ - Implemented by adding list with one element $O(1)$.

Remark. In the definition of the problem we choose our elements names to be a natural numbers in a known range. We can use a general names (i.e strings). In order to do that we are going to use a data structure which convert effecintly betweennames and natural number E.g hash table.

Remark. In the first (and in the following implementation) we assumed that given number of elements n in advance, hence, we could assume existance of a big array in size n to our implementation. If the case wasn't like that we are forced to implement a dynamic array.

In the first naive implmenetation.

(1) $Makeset$ in $O(1)$.
(2) $Find$ in $O(1)$.
(3) $Union$ in $O(1)$.

In the second naive implmenetation.

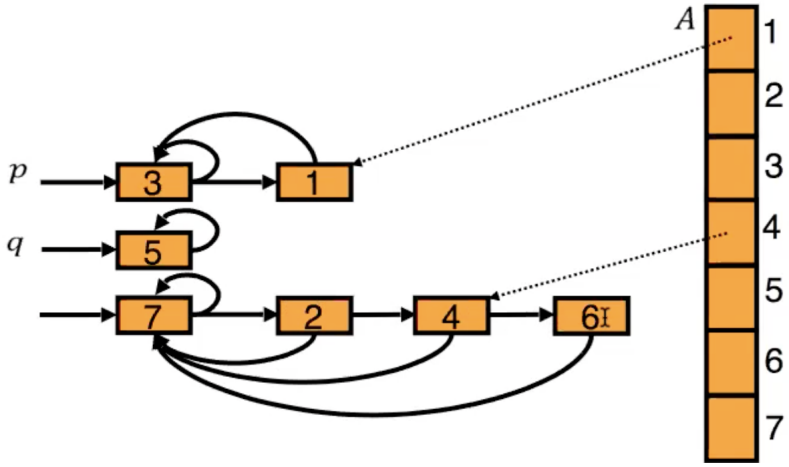
(1) $Makeset$ in $O(1)$.
(2) $Find$ in $O(n)$.
(3) $Union$ in $O(1)$.

We will suggest a solution which has.

(1) $Makeset$ in $O(1)$.
(2) $Find$ in $O(1)$.
(3) $Union$ in $O(logn)$ amortized time.

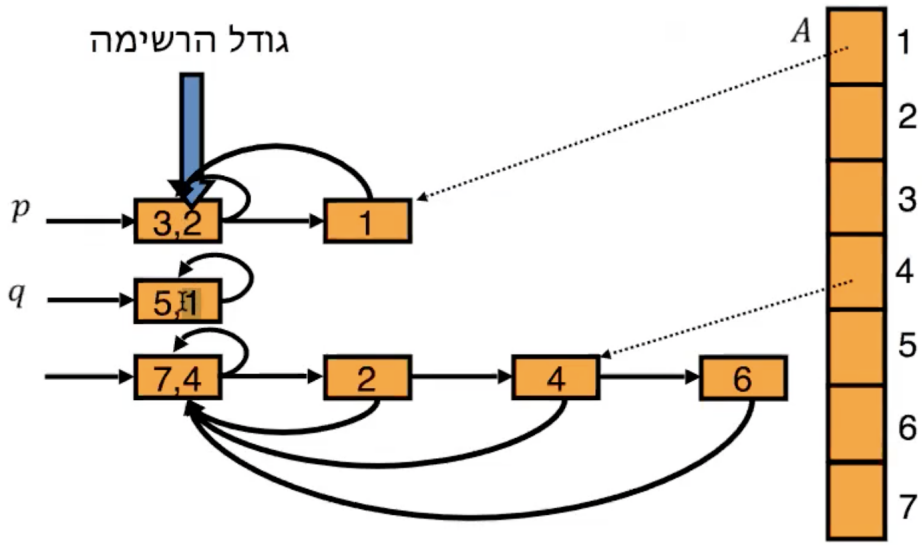
In the end we will suggest a data structure in which the amortized time of the operation $Union, Find$ is "almost constant" and the single operation time of $Union$ is $O(1)$..

Third implementation. We will represent each set as a list. Every element in the list point to the head of the list and to the next element in the list. We will hold a array which map elements $A[i]$ and in which point to the element i. A set is represented as pointer to the it list head.



- $Makeset(i)$ - is implemented by creating new node which point to itself and $A[i]$ - $O(1)$.
- $Find(i)$ - implemented by walking on $A[i]$ and return a pointer to the list head - $O(1)$.
- $Union(p, q)$ - implemented by union of lists which are pointed by p, q to one list and updating the elements pointer to our new list head. The function return the head of the new list $O(n)$.

We will represent rach set as a list. Moreover, in the list head we will hold variable which tell us it size. *Find* we be executed as before. *Union* we be executed by adding the small set to the bigger set. I.e the new list head will be our bigger set head, and all the pointer will point to it.



How are we going to do $r = Union(p, q)$?.

Definition. if m operations are executied in M time, then the amortized time for each operation is defined $\frac{M}{m}$.

Motivation for definition. Long sequence of operation of operation which almost of them are easy to execute (e.g $find$ in the last implementation) and some of the operation is hard to execute (e.g *union* in the same implementatio), so the efficiency of expensive operation devided for all the operation sequence. Moreover, maybe the execution time of operation decline by the "help" of previous cheap operation then the amortized time is scale which calculate the suppport of one operation to other.

Lemma. *If in each union we add the small set to the bigger then every element* x *change the set belong to at most* $log_2 n$ *times when* n *is number of elements in the structure.*

Proof. In each time x change set in which x belong to, then x move to new set which is bigger at least 2 time bigger than the current set. Since, at most $log_2 n$ transits the new set which obtain n elements and the belongs of our element x will not be change anymore. □

Lema motivation. E.g if we look at the following disjoint sets $s_1, ..., s_n$ *when* $s_i = \{a_i\}$ *then we have that* $a_1 \in \{s_1\}, ..., a_n \in \{s_n\}$ *now first we notice that if we look at* a_1 *and we union* $s_1 \cup s_2$ *i.e we add the little set to the bigger 1. Hence, now* s_1 *is no longer exists moreover, we have that* $a_1 \in s_2$ *i.e* a_1 *changed set one time now in order to insure one more change we need to build bigger set that* s_2 *assuming* s_3 *in order to add* s_2 *to it then we got more one change and we can do that* $log_2 n$ *at most.*

Lemma. *(Powerful wooding). if in union of sets n add the smallest set to bigger, then for each element xwhich never change the set he belongs to, he change the*

set which he belong to, at most $log_2 n$ times when n is number of elements in the structure.

Theorem. *if in each union we add that smallrst set to the bigger them the time of executing sequence of m operations Union/Find/Create is at most $O(mlogn)$.*

Proof. Notice that:

- Each operation of find or create take $O(1)$.
- Every union operation take $O(1)$ and additional time which linear depend on number of element which change set in the structure.
- By lema, for m operations there is at most $nlogn$ changes of set (for some elements). Each change cost $O(1)$.
- So the total time is $O(m + nlogn)$
- Since number of element n is less than number of operations m (we need create for each element in the structure), the the required time is $O(mlogn)$.

□

Corollary. *Every Union/Find/Create operation take $O(logn)$ amortized time.*

Theorem. *if in every union we add the small set to the bigger then the total time for executing a squence of m operations of $Union$ is at most $O(mlogn)$.*

Proof. (The difference from before that we can't say that $m < n$ since m operation include *create*).

- As before, each union operatin cost $O(1)$ and some time which s linearly depend on the number of elements which change the sets they belong tp in union.
- By the lema, for m sequence of m union operation, if there is t elements which there set change, then at most $tlogn$ sets change. Every change execute in $O(1)$.
- At total the required time for executing m operations of $Union$ is $O(m + tlogn)$.

Lemma. *lema: each element which is not the head of set change it set by previous union operation.*

Proof. Notice that each which is head in the smaller set change it set E.g if wee take $\{5\}, \{1, 3\}$ now we add $s_1 = \{5\}$ to $s_2 = \{1, 3\}$ now 5 change it set to s_2 no we have $s_2 = \{1, 3, 5\}$ in order to add it to s_3 there is requirement that $|s_3| > |s_2|$ then our head which was 1 in s_2 will change it set. Now we can do that in induction. Assuming we have t elements in the structure and we do m operations. Now we prove in induction.

Base. for $m = 1, n = 1$ and number of sets in the structure is $0 < k \leq n$ we get that $k \leq 1 \to k = 1$ then we get that the element after one union operation we have that it didn't change set I.e $t = 0 \leq m$..

Step. Assume that we have $m - 1, n - 1$ and $k \leq n - 1$ element in the structure then satisfied that $t - 1 \leq m - 1$..

Proof for $n \in \mathbb{N}$..

Now we want to proof for $n \in \mathbb{N}$. Assuming now we do m union operations and we know that for $m-1$ satisfied that $t \leq m-1$ when $t-1$ denote number of elements which change set after $m-1$ operations. Now Since we have $k \leq n$ we get that there is k sets in the structure $s_1, ..., s_k$ now assuming we union two sets s_i, s_j WLOG $|s_i| < |s_j|$ then the head of s_i then we have now t and m opertation Moreover, $t-1 \leq m-1$. Now using induction step we get $t = (t-1)+1 \leq (m-1)+1 = m$.. □

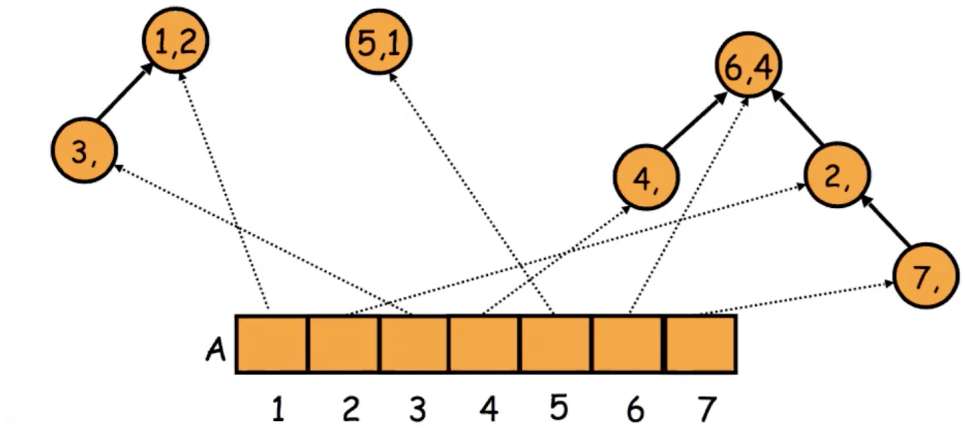
□

Corollary. *there is at most one element which has change in the first time: The element which is head in the smallest set.*

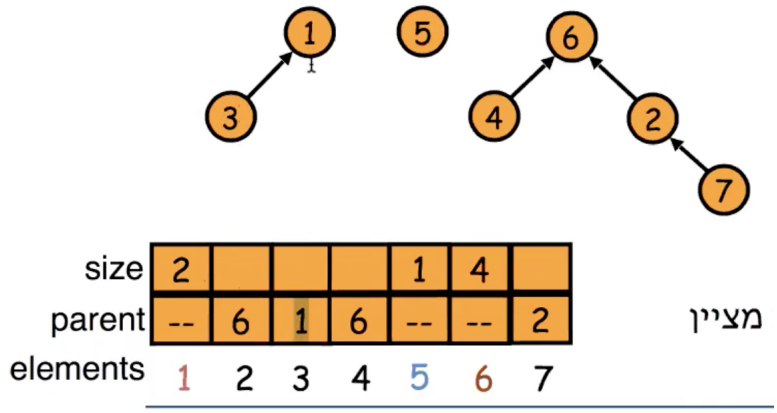
- *I.e, number of elements which change set is less that number of union m hence, $t \leq m$.*
  - *Since $t \leq m$ the time to exeucte m operarions of Union is $O(m + tlogn) = O(mlogn)$.*

fourth implementation. For every set we create inverse tree (childs point to parent). And node to each element in the set. The tree root will also hold number of elements in the tree. Moreover, we will hold access array as in the third implementation. In *Union* operation we pout the root of the the smallesr tree under the tree root of the biggest tree and update the size of the tree.

Example. The sets $\{1, 3\}, \{5\}, \{2, 4, 6, 7\}$.



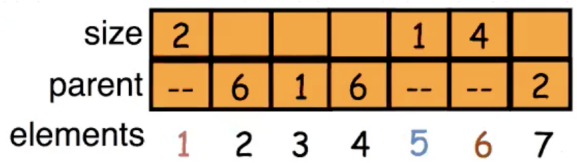
In assumption the number of elements in known in advance, we can represent all the lists in array without using pointers. We can do that in all the previous implementation.



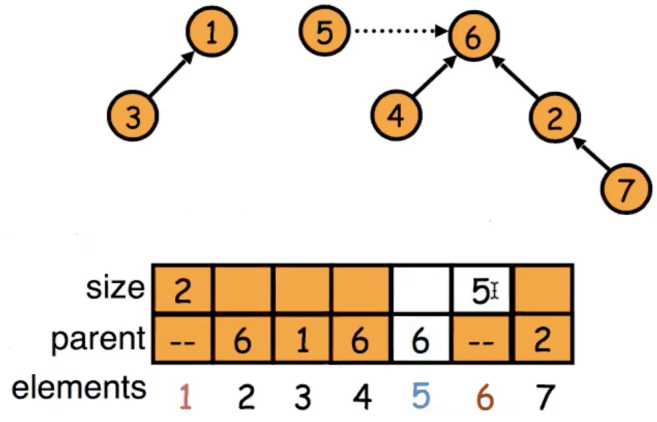
Implementation: the root element is indicator of a set.

- $Makeset(i)$ - implmented by creating tree with a single node i. Time $O(1)$.
- $Find(i)$ - implemented by tranversal on the tree.
- $Union(p, q)$ - implemented by union trees which there roots are p, q S.T the root of the smallest tree will be given as child to the root of the bigger tree. $O(1)$.

Example. The intial case as in previous.



After $Union(5, 6)$ we get that:



The time required of $Union$ is $O(1)$.

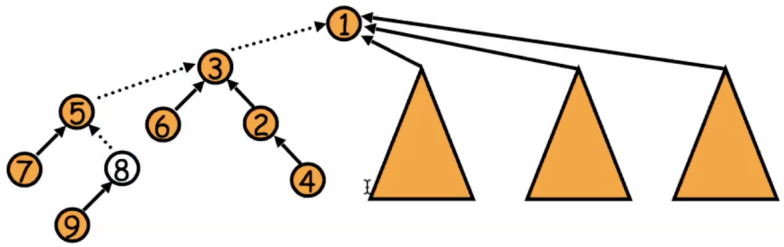
Lemma. *for a forest of inverted trees with n nodes and height h which is made of unions of smaller sets into bigger sets. Satsify $h \leq log_2 n$.*

Proof. We remember the first lema A for representing sets using lists and union by size. If in each union we add the smaller set to the bigger then for each element x

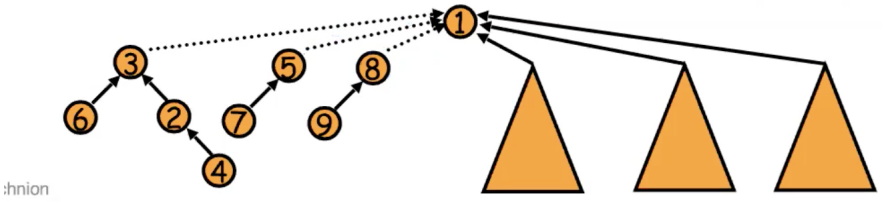
change the set which he belongs to at most $log_2 n$ times when n number of elemets in the structure. Since each set change lead to increase the distance of the element from the root by 1 and each element start with distance 0 from it's sets root (each element start as a set of single element) Hence, the distance of node from it's root is at most $log_2 n$. □

Corollary. *The time for Find operation is $O(logn)$ in the worst case when n is number of nodes in the trees.*

When we execute $find(i)$. We update the parent of each node from i to the root, in a way that they all point to the root. E,g $Find(8)$.



The implementation require two tranversals to the root. The first is to find the root, and in the second is to update the pointer to point to the roots. From here the complexity of $Find$ in the worst case still $O(h) = O(logn)$.



Theorem. *In implementation using collection of tree with n elements with union by size and compress pathes, the time complexity of executing m operations of $Union/Find$ i bounded by $O(mlogn)$. Hencem the amortized time for each operation i bouned by $O(logn)$.*

Exercise. In a dark room there is n ball which are numbered by $1, 2.., n$. Every ball is colored with some color, but we can't see it's color. Known that the is only two possible color. Implement data structure which support the following operations.

- $Init(n)$ - the room is initalized with n colors which there color not known.
- $Same(i, j)$ - will till if the balls i, j are in the same color.
- $Different(i, j)$ - will till if the balls i, j if they are in different colors.
- $AddSameCoor(i, j)$ - Will anwer the quetion whether balls i, j are in the same color? if yes then it returns true, if no then it returns false, otherwise if it can't determinne it return $UNKNOWN$.

Proposition. *Expect of the opertaion $Different$ we can solve the problem using $Union - Find$, when our sets is the balls which are known that they have the same color. In order to take care of the addition function we see that if $color(i) \neq color(j)$ and $color(i) \neq color(k)$ then $color(k) = color(j)$.*

Init(5)	
Different(1,2)	
Different(3,4)	
AreSameColor(1,3)	UNKNOWN .
Same(1,3)	
AreSameColor(1,3)	TRUE .
AreSameColor(2,4)	TRUE .
Different(4,5)	
AreSameColor(3,5)	TRUE .

Notice that:

$$color(1) \neq color(2), color(3) \neq color(4), color(1) = color(3)$$

Since there is 2 color we deduce $color(2) = color(4)$. Hence, we get that $AreSameColor(2,4) = TRUE$. Now of the last operation we see that: $color(3) \neq color(4) \neq color(5)$ we get $color(3) = color(5)$.

Solution. We will hold disjoint sets of balls, when each set will save a pointer to a set of balls which are known that they are with different color. This pointer is called $diff$;(COMPLET LATER).

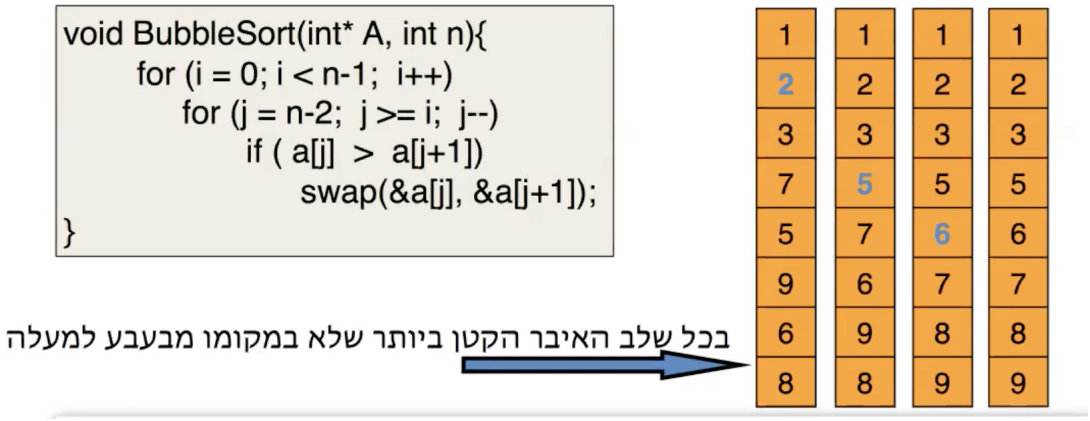
Heap sort, Quick sort.

Input: array with n elements.

Output: array in which the elements are stored in a increain way.

A naive sorting method requires $\Theta(n^2)$. E.g chaging sequential (Bubble sort).

```
void BubbleSort(int* A, int n){
      for (i = 0; i < n-1;  i++)
          for (j = n-2;  j >= i;  j--)
               if ( a[j]  >  a[j+1])
                    swap(&a[j], &a[j+1]);
}
```



Remark. In general we define lists by their keys and we don't imulsively sort. We assume that for each key there is data which point to the place of other information.

Other algorithm is merging sort, in cambinatorics algorithm course we saw it.

Heap sort.

Definition. Heap is a data structure which define by the following operation.

- $MakeHeap(Q)$ - create a empty heap.
- $Insert(x, Q)$ - insert x to the heap.
- $Max(Q)$ - print the value of the most bigger key in the heap.
- $del\_max(Q)$ - delete the value of the bigger key in the heap.

Abstract. The most using for heap is a priority queue there we add values, and each time we delete the value with the biggest key in queue. E.g every value could define a job with priotity degree. Our next mission is to execute the jon with the highest priority. Other use for heap is sorting.

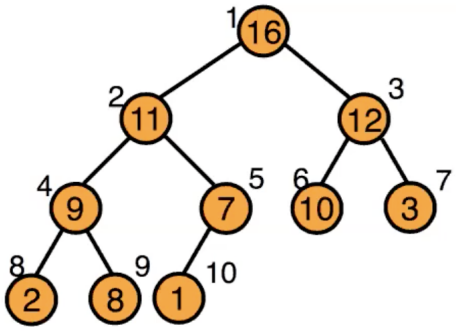
Naive implementaiton for heap. A binary search tree with insertion, deleting, and find maximum in $O(logn)$..

Sorting using heap.

- HeapSort - sorting using heap,
- .For each x in the input. $Insety(x, Q)$.
- As long as the heap is not empty:
  - $Output\ max(Q)$
  - $del\_max(Q)$

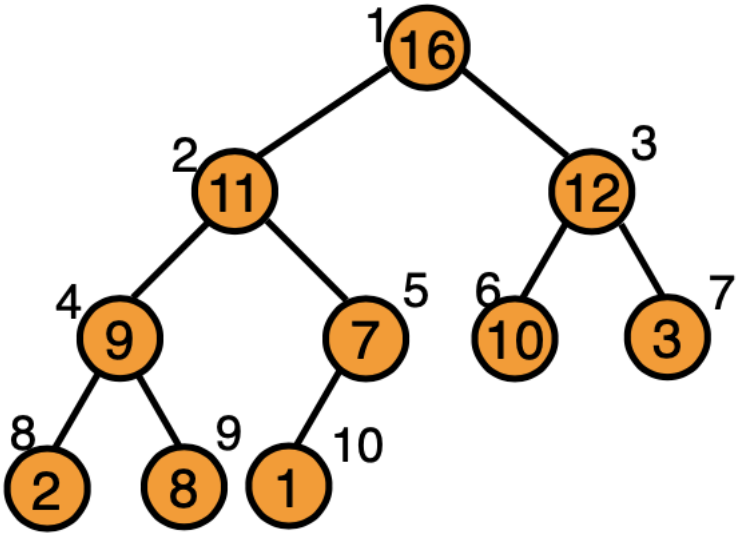
Remark. In the standard implementation for heap let as find quickly the maximum (In constant time) and define the operation $make\_heap(Q, x_1, ..., x_n)$ which creaete a heap with n elements in complexity og $O(n)$. For binart search tree, creating the heap from n elements take time of $\Theta(nlogn)$.

Standard implementation. Implemeting heap using a binary tree almost complete (which is not searching tree). The key of parent is bigger equal then the keys of it's children (This condition is called the heap property). The order between children is not limited. E.g.



As remembered in the last lectures, a almost complete tree represented in array. When the parent i is in the node $\lfloor \frac{i}{2} \rfloor$, a left child of node i is node $2i$ and right child is $2i + 1$.

Standard implementation of heap. We can implement a heap using a almost complete binary tree (not searching tree). The key of parent is bigger than all childs keys (this condition is called heap property).The order between childs is not determined. E.g:

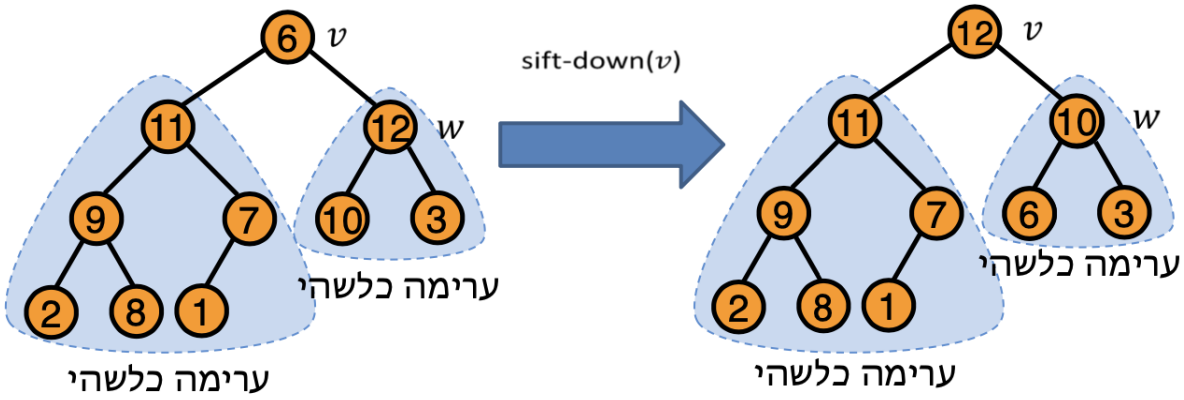


Advanatages:

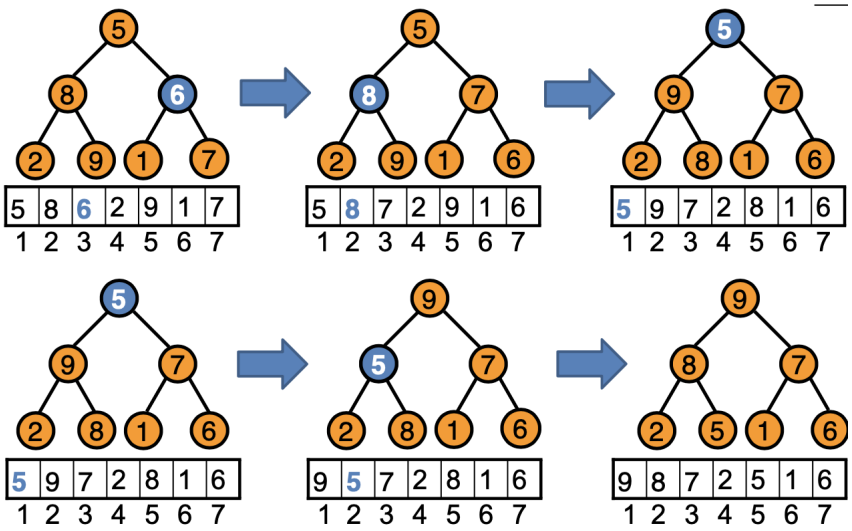
- We can get the max in $O(1)$.
- It's easier to to balance, (not searching tree).
- We can create heap fast.

Sift-Down

In time of inserting and deltinh we need to preserve the heap property. In order to do that we are going to use sift down procedure. This procedure invert a binary tree in which the property were damaged only in the heap root. Our tree is made of root v which point to two heaps.



Make-Heap prodecure. Implementing $Make\_heap(t)$: for all treeminterior nodes s.t we pass on the children before it's parent. (E.g postorder). For each node in the tree we do $sift\_down(v)$ E.g:



Remark. The last array not soerted but satisfy the heap property.

Correctness. In time our algorithm execute $sift - down(v)$ the node v will point to a tree which satisfy the heap property. Hence, after doing $sift - down(v)$ we will get heap which it's root is v. This is true for every node, including the root. Therefore, in the end we get heap.

Time running of make-heap. Given by O(sum of heights for all nodes), We can run example which show that. For each node we execute $O(h(v))$ fixes when $h(v)$ is node v height. A naive analyze also give a bounder of $O(nlogn)$ when each node height is bounded by $O(logn)$. A more accurate analyze will take into account that most of node got a small height, Analyze like that will give us bounder of $O(n)$.

Time analayziz of make-heap prodecure. The running time of make-heap given by sum of tree nodes height $O(\sum_v h(v))$.

Theorem. *For a complete binary tree in height h which obrain $n = 2^{h+1} - 1$ nodes, the sum of heigts is less than n.*

Corollary. *Running time of make-heap on $n$elements in $O(n)$.*

Proof. There is only one node in height h, and 2 nodes in height $h-1$, 2^2 in height $h-2$ and in general we see 2^i in height of $h-i$. Hence, sum of height which denoted by S satisfy:

$$S = \sum_{i=0}^{h-1} 2^i(h-i) = 1 \cdot h + 2(h-1) + 4(h-2) + ... + 2^{h-1} \cdot 1$$

$$2S = 2 \cdot h + 4(h-1) + 8(h-2) + .. + 2^h \cdot 1$$

$$\Rightarrow S = -h + 2 + 4 + 8 + ... + 2^h = -h - 1 + \overbrace{(2^{h+1} - 1)}^{n} < n$$

□

Remark. What happen when our tree is not full tree with n nodes?

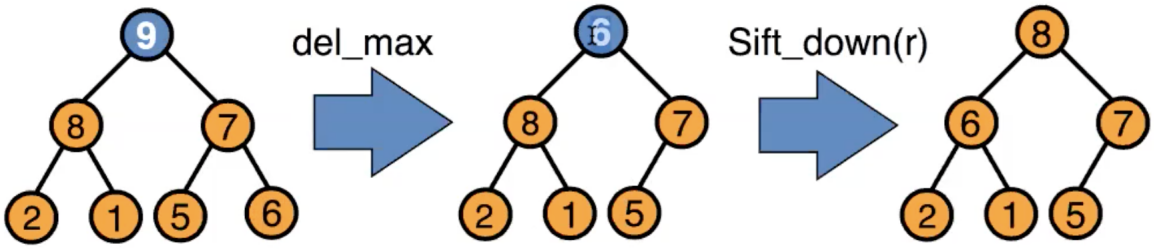
We will complete it to full tree. At most we add n nodes. Sum of height in the uncomplete tree is less then sum of heights in the full tree which is less than $2n$. I.e, sum of heights is less than $2n$. Hence, the running time of make-heap is also $O(n)$ also for non complete tree.

Implementing operations and sorting.

How are we going to implement $max(S)$? BY Printing the root. $O(1)$ time complexity.

How $del\_max(S)$ implemented? Replace the most right (or other node in the last level) with root r. Execute $sift\_down(r)$. Time complexity $O(logn)$.

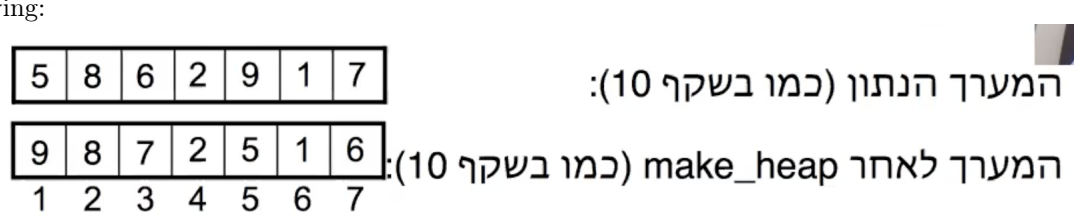
Example. As following:



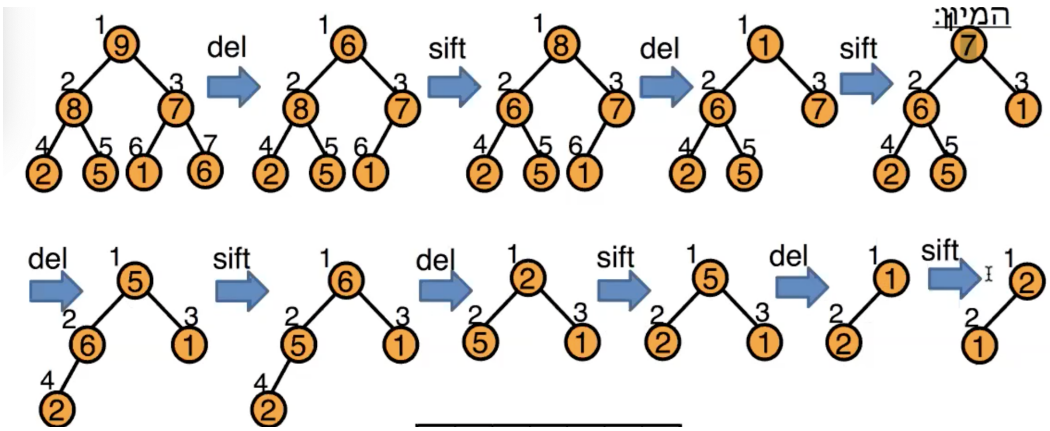
Heap sort with using make heap.

```
HeapSort(x_1, ..., x_n){
    make_heap(S, x_1, ..., x_n);
    While ( S ≠ ∅) {
        Output max(S);
        del_max(S);
    }
}
```

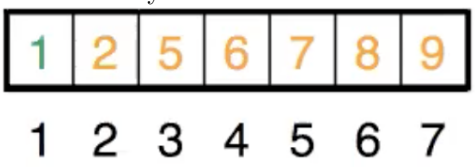
Hence, we have algoeithm to sort a array by making heap and getting the following:



The soerting operations are as following:



The sorted array is:




```
heap_sort(n){
        for (int i = n/2; i > 0; i--)      /* בנית הערימה */
                sift_down(i,n));
        for (int i = n; i > 0;  i--){      /* המיון עצמו */
                swap(1,i);
                sift_down(1,i-1);
        }
}
```

The sorting code.

Remark. We can also sort from bigger element to smallest.

Time complexity. The total time required of making heap is $O(n)$ and n operation of del_max i.e the total time is:

$$O(n) + O(nlogn) = O(nlogn)$$

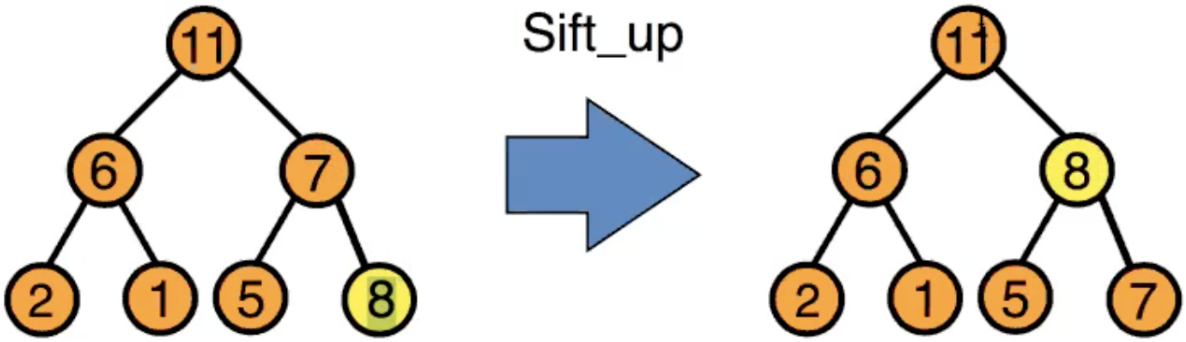
Advantages of heap sort algorithm for sorting.

- $O(nlogn)$ complexity for worst case.
- No need for helping space.

Inserting element ro heap (in case user wanted to make bigger array from the current).

- Creating new leaf in the first free place.

- We insert to it the wanted value.
- Now our heap property damages exactly in the leaf place.
- We execute sift up till the new value go up to it's right place.



Quick Sort.

The sorting rely on devide and conquer.

- In each step we sort the array $A[l,..,r]$
- We will choose a element from $A[l,..,r]$ which are called the pivot and recalled by $A[p]$. E.g we choose $p = l$ and we will choose p randomly.
- Devid: order $A[l,..,r]$ s.t all elements less than pivot will be in $A[l,..,i-1]$ and all the bigger element will be foind in cells $A[i,..,r]$ when the pivot is at $A[i]$.
- conquer: sort recursively the arrays $A[l,..,i-1]$, $A[i+1,..,r]$

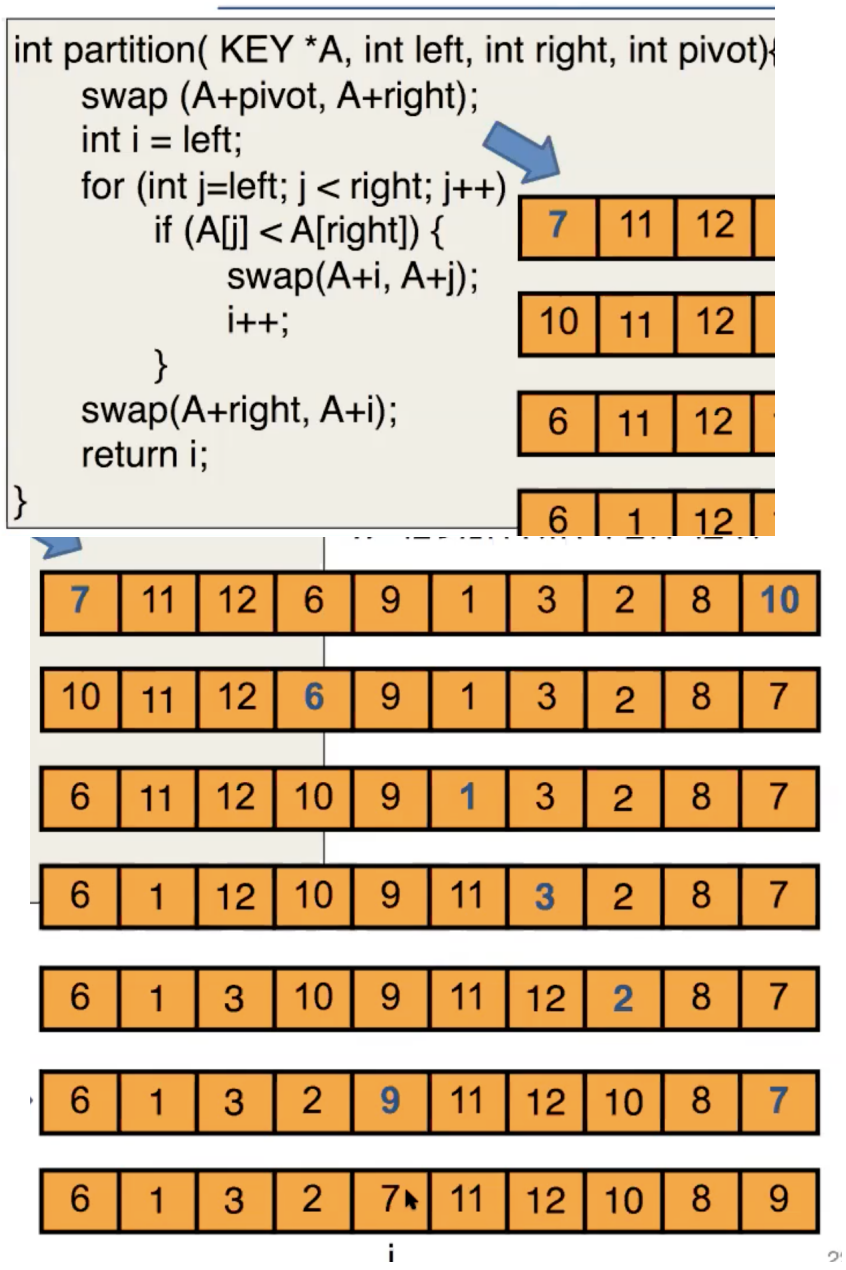
```
QuickSort(Key *A, int l, int r){
    if (l >= r) return ;
    int p = choose_pivot(A,l,r);
    int i = partition(A,l,r,p);
    QuickSort(A, l, i-1);
    QuickSort(A,i+1,r);
}
```

Partition return the index of pivot.

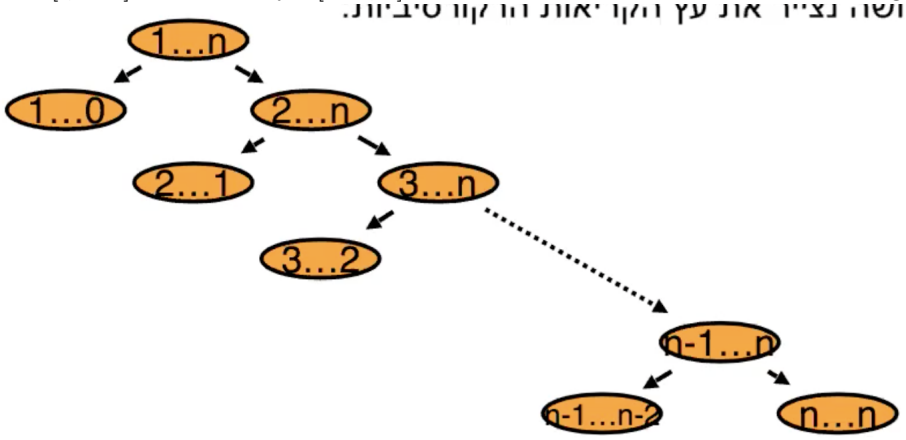
Coreectness. The pivot exist in it's right place in the array after partition.. The recursive calls will sort the element before pivot and after pivot so we get sorted array.

Running time rely on the pivot element.

(1) In the worst case $\Theta(n^2)$.
(2) If in each step the array is devided in equal way. Then we get time of $O(nlogn)$.
(3) If our pivot chosen randomly, we get that the running time is $O(nlogn)$ in average. I.e number if itirations of this algorithm doesn't stem from our maximal run time.



Given a sorted array from small to bigger element. We assume that the element which we choose is the smallest. In each call recursive on $A[l,n]$ executed a recusrive call to $A[l,l-1]$ and the array $A[l+1,n]$. So our recurisve calls will look as following:



Running time.

- Worst case.
  –

$$T(n) = cn + T(n-1) = cn + c(n-1) + T(n-1)$$

$$= c\sum_{i=1}^{n} i = \Theta(n^2)$$

  So it the worst time running.
- Optimal case: When put pivot is median.

$$T(n) = 2T(\frac{n}{2}) + \Theta(n)$$

$$T(n) = \Theta(nlogn)$$

- Balanced partiotion: Our pivot devide the array in a way that a part will contain some precentage of the array elements (E.g 10%).

$$T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + \Theta(n)$$

$$T(n) = \Theta(nlogn)$$

- The expectation time running for a random pivot is:

$$T(n) = \Theta(n) + \frac{1}{n}(\sum_{i=1}^{n} T(i-1) + T(n-i))$$

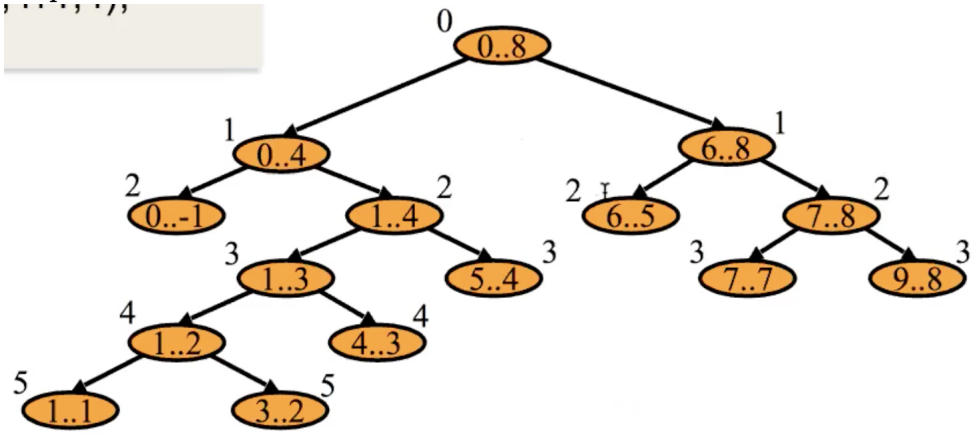
$$T(n) = \Theta(nlogn)$$

  Our recursive call is a average binart tree. The recusive formula is identical to the formula of bulding a random binary search tree and from here the expectation running time of quick sort for random pivot is $O(nlogn)$.

Space complexity. The space required to QuickSort include.

- A input array $O(n)$.
- Recursion stack (storing $O(1)$ local variables in each level).
- In the worst case the recusrsion depth is $O(n)$.
- Asymptotic it's not a problem, but in reality it's expensive in time and space.
- We will see know a method to limit the recusrion depth to $O(logn)$ in the worst case.

Example. Recursion stack behaviour.



In each node written our index range, and the number beside it mark our current depth. So our method is calling in loop insted of calling recursively, since calling recursively allocate a space in the stack which is less preferred so this is called tail recusrion. Some compilers do it automatically to make program run more efficient.

```
QuickSort(Key *a, int l, int r) {
    if (l >= r) return ;
    int p = choose_pivot(a,l,r);
    int i = partition(a,l,r,p);
    QuickSort(a, l, i-1);
    QuickSort(a, i+1, r);
}
```

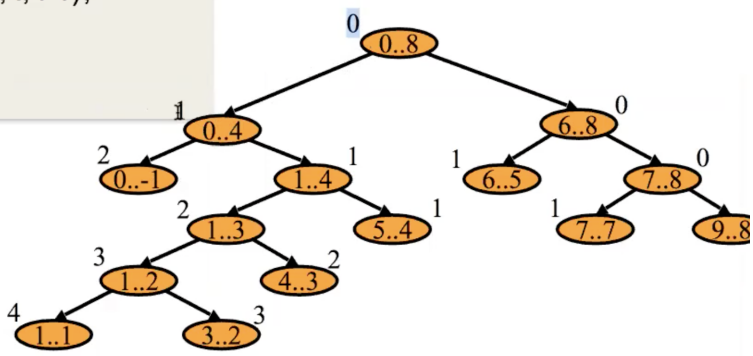
```
QuickSort(Key *a, int l, int r) {
    while (l < r){
        int p = choose_pivot(a,l,r);
        int i = partition(a,l,r,p);
        QuickSort(a, l, i-1);
        l = i + 1;
    }
}
```

This codes are equivalence, but the main different is that the code in the right rely on calling in loop which won't allocate space in stack but the left code do that. First we see that our code work as following, we choose a random pivot, and we execute partition so all the bigger eleemt will appear in right and smaller will in left our pivot. And our pivot is in index i so now we call recucrsively, for two sub arrays $A[i+1,r], A[l,i-1]$ this calling are expensive in our stack. So instead we are going

to use the loop and avoid addition recusrive calls. So first what are we going to do is choosing a pivot, not our pivot left to it is called recusively now in the right side find pivot to it then now our right side is devided to two parts onewill be called recuively as in the beginning and the right by loop, we keep doint that till we get soerted array, bot know our recusion depth will be more less since calling right subarray won't cost space in the stack. Now we can also do that for right side i.e our right side could be called by loop and right in recusrion.

```
QuickSort(Key *a, int l, int r) {
    while (l < r){
        int p = choose_pivot(a,l,r);
        int i = partition(a,l,r,p);
        QuickSort(a, l, i-1);
        l = i + 1;
    }
}
```

החלפת הקריאה הרקורסיבית
השניה באיטרציה:

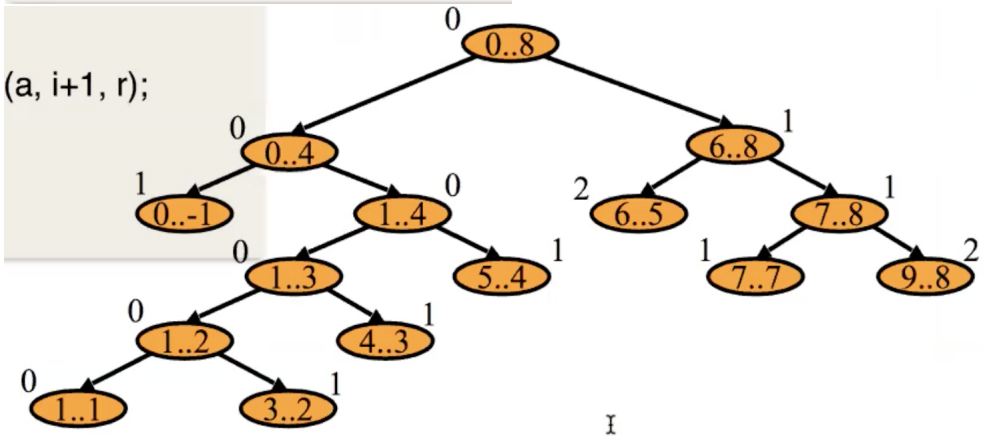


as we said the first call could be replaced by calling in loop and second in recursion, the following code illustrate that:

```
QuickSort(Key *a, int l, int r) {
    while (l < r){
        int p = choose_pivot(a,l,r);
        int i = partition(a,l,r,p);
        QuickSort(a, i+1, r);
        r = i - 1;
    }
}
```

More efficicient method rely on above. We can call recusively the smallest subarray and in loop the biggest, this way we could use less space in the stack.

```
QuickSort(Key *a, int l, int r) {
    while (l < r){
        int p = choose_pivot(a,l,r);
        int i = partition(a,l,r,p);
        if (i - l < r – i) {
            QuickSort(a, l, i-1);
            l = i + 1;
        }
        else {
            QuickSort(a, i+1, r);
            r = i - 1 ;
        }
    }
}
```



In this implementation our recusin is bounded by $log_2(n)$ since in each call our array is less by half. Notice that in a case we choose every time the pivot at beginning we get that our recustion depth is $O(1)$ since we always go right and it cost 0 in depth. While before it cost us expensive space of $O(n)$.

Hashing exercises.

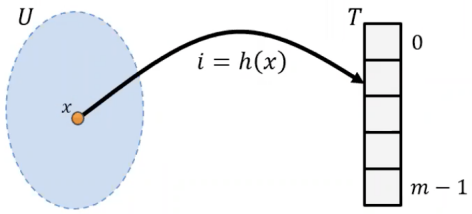
Requireda data structure which support the operations $Insert(x), Delete(x), Memeber(x)$ for n people which are identified by id.

Complexity required. *Time* $O(1), Space : O(1)$..

- Solution with array is not compatible beacuse of space complexity.
- Solution by BST is not compatible because of time complexity $\Omega(logn)$ in the worst and average case.
- We can compromis by average time of $O(1)$.

General problem. Given that our world is set of elements U and set Kwth size of n of values from U. Given that $|U| > |K|$. Required a data structure with efficient implementation for the following operations with space complexity of $O(n)$.

Rehashing definition. We will allcoate a array T with size m when $n = O(m)$. We will define a hashing function $h : U :\to \{0...., m-1\}$ for each element $x \in U$ the hash function will return the cell number in T in which we will store x.



Example. Returning cell for homewprks when:

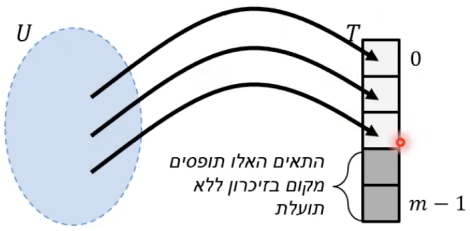
- $U-$ collection of all possible ID.
- $h-$ hash function $h(id) = idmod(100)$.

Some questions.

- How are we going to choose our hash function?
- What we do in case crash exists? (I.e if two values $x \neq y$ in which $h(x) = h(y)$.

Choosing a hash funciton, In hash function we need to satisfy the following requierments.

(1) Calculating time of $O(1)$.
(2) The function is surjective. (The domain of h is all indexes T).
(3) Uniform scatter of keys.



Load factor. The load factor is defined as $\alpha = \frac{n}{m}$..

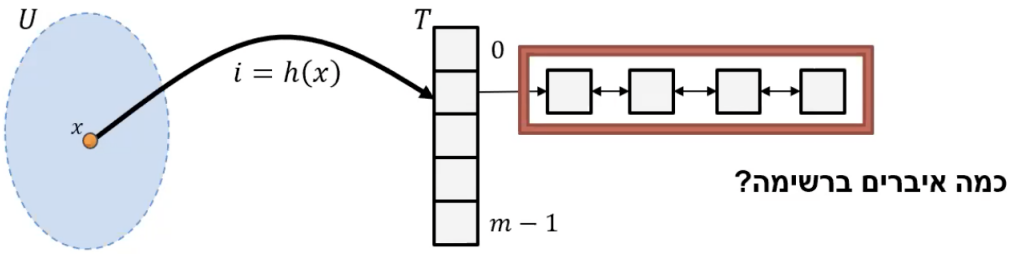
- n- is number of elements in the table.
- m - size of the table.

if satisfied that $n = O(m)$ then $\alpha = O(1)$.

The uniform scatter assumption. h scatter in uniform way between our table cells. I.e, number of average elements which map to i in the table is α. *E.g* $h(x) = xmodm$ *is a function which scatter in uniform way. We can look as a sequence of input which is bad for our scatter assumption for example, we can take* $m, 2m, 3m, .., ,$ *those inputs will be map to the same cell in the table.*

Solution for crashing.

Chain hashing. each cell in our table point tp a link list, if $x \in U$ were inserted to the structure, then it will be in the list which point to $T[h(x)]$.



Double hashing.
Open addressing. The elements are in a hash table (the different from chain hashing). The difficulty with dealing with crashes is that we can't store more tha one element in a cell. A mathod to deal with crashes in Open addressing is called Double hashing.
Double hasing. We will define a infinite sequence of functions

$$h_k(x) = (h(x) + k \cdot r(x)) mod m$$

when $h(x)$ is a hash function and $r(x)$ is a step function.

General idea. if a cell $h_k(x)$ is used, then try $h_{k+1}(x)$..

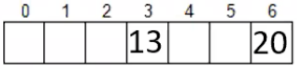
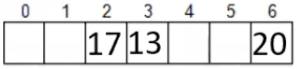
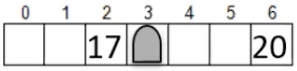
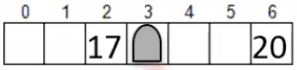
Example. $h(x) = xmod7, r(x) = 1 + (xmod5)$.

Reminder.

$$h_k(x) = (h(x) + k \cdot r(x)) mod m$$

.

(1) What is the size of the table?
(2) Discribe the table after those operations.

	$h(x)$	$r(x)$	$h_0(x)$	$h_1(x)$	$h_2(x)$	T
$Insert(20)$	6	1	6			
$Insert(13)$	6	4	6	3		
$Insert(17)$	3	3	3	6	2	
$Delete(13)$	6	4	6	3		
$Delete(17)$	3	3	3			

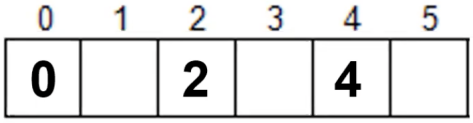
Requierments from the hash function and r in double hashing.

$h(x) = x \ mod \ 6$

$r(x) = 1 + (x \ mod \ 5)$

$h_k(x) = \big(h(x) + k \cdot r(x)\big) \ mod \ m$

$Insert(8)$



$h(8) = 8 \ mod \ 6 = 2$

$r(8) = 1 + (8 \ mod \ 5) = 4$

$h_0(8) = 8 \ mod \ 6 = 2$

$h_1(8) = (2 + 4) mod 6 = 0$

$h_2(8) = (2 + 2 \cdot 4) mod 6 = (10) mod 6 = 4$

$h_3(8) = (2 + 3 \cdot 4) mod 6 = (14) mod 6 = 2$

Target: The combination between $r(x)$ and $h(x)$ will insure us a pass on all the array cells in surjective way and a uniform scatter.

(1) We need to choose r like that, in which for all x satisfied $r(x) \neq 0$ and $r(x) < m$).

(2) We will choose m, r s.t for all x won't be common devidors. I.e, satisfy:

$$\forall x \in U, GCD(m, r(x)) = 1$$

E.g if $m = 20$, $r(x)$ shouldn't get the values $2, 4, 5, 6, 8, 10, 12, 14, 15, 16, 18, 20$.

Possible solution. In general we choose m to be prime, in order to satisfy condition 2. One way which we satisfy that other conditions is by choosing function in the following form.

$$h(x) = xmodm$$

.

$$r(x) = 1 + (xmod(m - c))$$

- *mod* could return 0, hence we add 1 to $r(x)$.
- The term $xmodm$ coud return $m - 1$ then we get that $r(x) = 0$.
- Hence, we do the calculations $xmod(m-c)$ when c is small positive integer.

Complexity. The complexity analysis of the following function show that the probability that our searching sequence will be small. Hence, the complexity of the three operations $Insert, Member, Delete$is $O(1)$ in average on the input. But $O(n)$ in the worst case.

Rehash. When we did Double hashing and removed elements we were required to mark places as free to insert and find which could complicate our structure, in order to solve this we do Rehash.

Definition. Rehash is moving all the table elements to a new table in identical size using our hash function. Time complexity of this operation is $O(m)$ in average on input? Why in average on input. If we do rehash one time in $\Omega(m)$ delete operations, then the amortized time complexity of delete will be $O(1)$ in average.

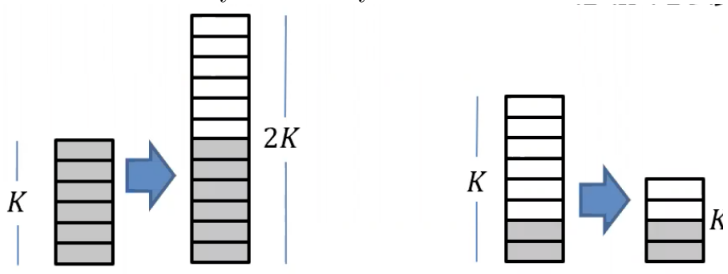
Example. $\frac{m}{2} = \Omega(m)$ for $c = 2$ we will de rehash in a cost of $O(m)$ as one time in $\frac{m}{2}$ delete operations hence, the amortized time complexity of delete will be $O(1)$ in average.

$$\underbrace{O(1), O(1), O(1), \dots, O(m)}_{1 \quad 2 \quad 3 \qquad \frac{m}{2}}$$

Dynamic hash tables. We used to to do rehash when number of elements in the array is very small from the size of possible elements space. Till now we assumed that the size of the table satisfy $n = O(m)$ so for that, we assumed that we know in advance a size order of the elements will ne in our structure. But as sum cases number of elements we need is bigger than our first evaluate.

- This could affect the average time complexity of $Insert, Delte, Member$.

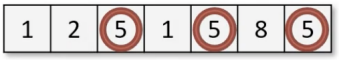
Solution. We will use dynamic arrays.



Under the assumption of a uniform scatter, the amortized time complexity of the operations $Insert, Delete$ will be $O(1)$ in average on the input.

Exercise. Find a algorithm which find a most remebered element in array with size n with $O(n)$ time complexity in average on input.

Example. In the following array:

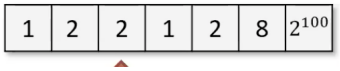


The element 5 is the most remembered element in the array.

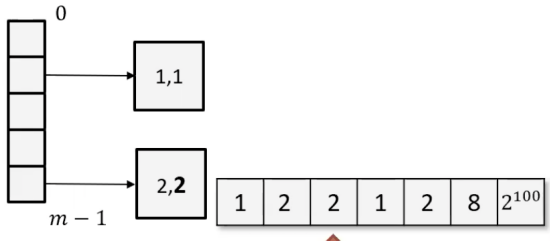
Naive solution.

- We will walk on the array and take the max element - $O(n)$.
- We will create array awith the maximal size and use it as histogra, - $O(n)$ (Intilizaing array within $O(1)$).
- When we fill the histogram we will will store using a variable which one the most recurrence. Like that, when we finish folling elements in the array we will know the most remembered element.

Problem. What if almost all of element are small and we have element which is large? why should we allocate table in it's size?



Good solution. A good solution is using a hash function on our histogram when hash table is histogram. E.g hash function $h(x) = xmodm$. This is a hash function with a uniform scatter for a random input.



for the following array we walk and see 1 then we do $h(1)$ and get a cell in out hash table, in the cell we point to a node which will store the element and number of reccurence to it. And we keep doing that till we reach the end of the array. Then we walk on it in $O(n)$ and take the most remembered element.

Average on input vs probalistic average.

Average on input: we assume that our input is random. E.g the probability to find a student in the course with specific id is $\frac{1}{10^9}$ *if id uniform distribute.*

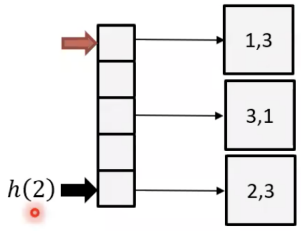
Practical: if we use a function with a uniform scatter (e.g modulo) then there will be a uniform scatter in the hash table we create.

probalistic average. We don't have any assumption on the input. E.g, assuming the size of table is m, in case there is a malicious secretary which add to the course only a student which t here id mod m give 0..

Practical: we can't use a regualar hash function anymore, so we can put the worst input. Hence, we use universal hash function as mentioned before.

Exercise. Discribe a algorithm which given array in size n including intergers, and positive number k, answer the question: if exists i, S.T number of recurrence of i and number of recurrences of $i+1$ is k in $O(n)$ time complexity in average on input.

Solution. We will use the histogram and hash function as previous question. Each cell in the histogram will store the number of recurrences of that elements in the array in a node which $h(i)$ point to it (chain hash). Now in order to do that we do as following: $k = 4$



First we have a point to the first element on the table which is 1. Then we do $h(1+1)$ and check the number of recurrence in $h(2)$ is in this case us $3+3=6\neq 4$ so we continue to the next element which is 3 it has recurrence of 1 and $h(4)$ has no recurrence so we get that $3+0=3\neq 4$ (note that for 4 we didn't create any node in the histogram for 4 we will not find it so we return 0 in this case since $h(4)$ will give us a cell which doesn't point to node that contain number of recurrence of 4) so we continue to the element 2 and see that $h(2)$ has a node with element 2 and reccurence of 3 then we look at $h(3)$ which point to a node with element 3 and reccurence of 1 so $3+1=4=k$ as wanted.

Remark. The question given with average on input that why $h(x) = xmodm$ work fine. But in probablilistic average we should use a universal hash function, which has no any previous assumption on the input, and could work for malicious users.

Union find exercises.

Structure definition. Gicen n elements $1, 2, ..., n$. Those elements are devided during running the program to disjoint sets. In the beginning of running, all the elements are devided into set in size 1 (Singleton).

The structure support two operations.

- $Find(i)$ - return the name of the set in which i element belong to.
- $Union(p, q)$ - Union both sets defined by p, q and return the name of the new set. After this operation, the sets p, q destroyed and instead we get $p \cup q$.

Implementation of Union find using inverse trees.

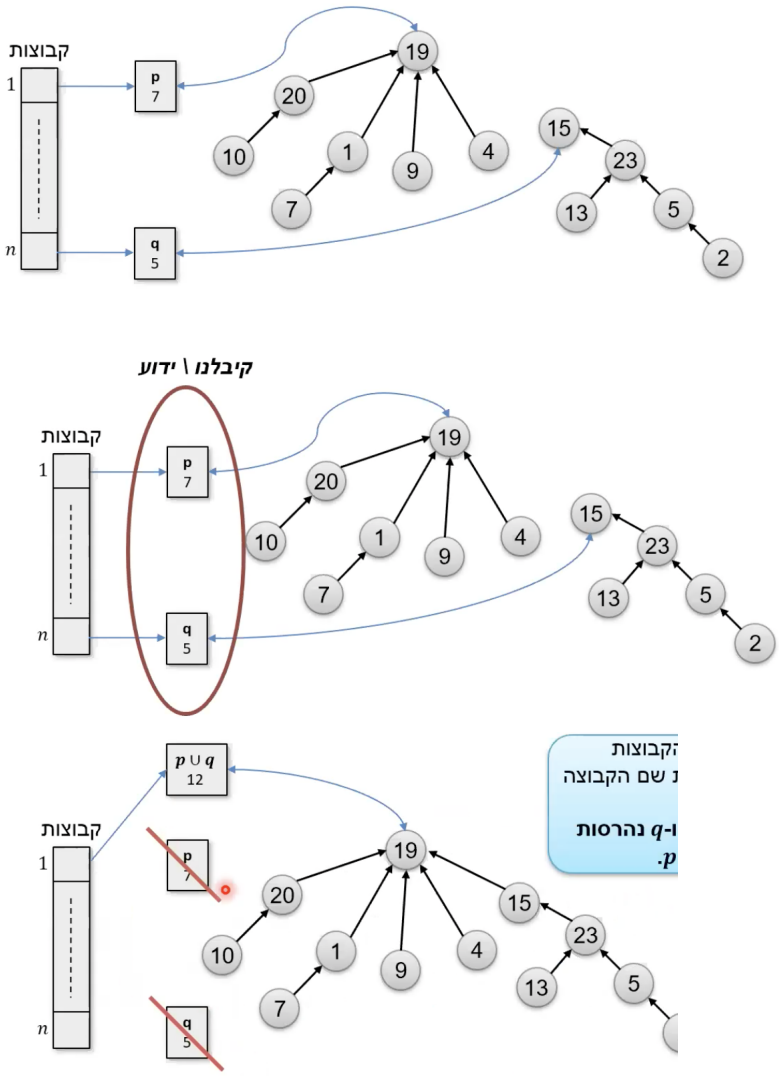
- For each set we create a invert tree (a tree which there childs point to parents) from all set elements.
- The root of each tree will point to a node which contain the set name and number of element in it
- In addition we will hold two arrays:

– Elements array: map between a element to a matching element in it's inverted tree.

– Sets array: map between numer of sets to the node which match the

Implementation of union.

- The sets array got access to both trees.
- We will let the first point in tree 1 point to the root of the second set.
- We will update the size of the new set and omit the set title from the sets array.



Implementing find operation.

- From the elements array we access the node in the inverse tree.
- We will go up in the inverse tree till we reach the root. From him we will access the node which has our set name and return it.

Time complexity: $O(n)$ in the worst case. Without union by size, the searching path may be n. We will see how union by size will improve it.

Improve 1 - union by size. If we de union by size, each find operation will take $O(log(n))$ in the worst case.

Proof. For every element i in the structure:

- The first distance of i from the set root is 0, since i is the only element in the set.
- Each time the distance of i from the root is bigger by 1, the size of set in which i exists is bigger by 2.
- The maximal set size is n and the distance of all element from the root couldn't be more than $log(n)$.

□

Impove 2 - compress pathes. After finding the element i, we will scan again the path between i to the root and we will connect every node in a path to root.

Theorem. *If we use union by size and compress pathes, the time complexity of m operations of $Find, Union$ will cost $O(mlog^*n)$. Hence, the amortized complexity of two operations together is $O(log^*n)$.*

Definition. log^*n is defined as following:

$$log^*n = min\{i | \underbrace{loglogloglog...log}_{\text{i times}}(n) \leq 1\}$$

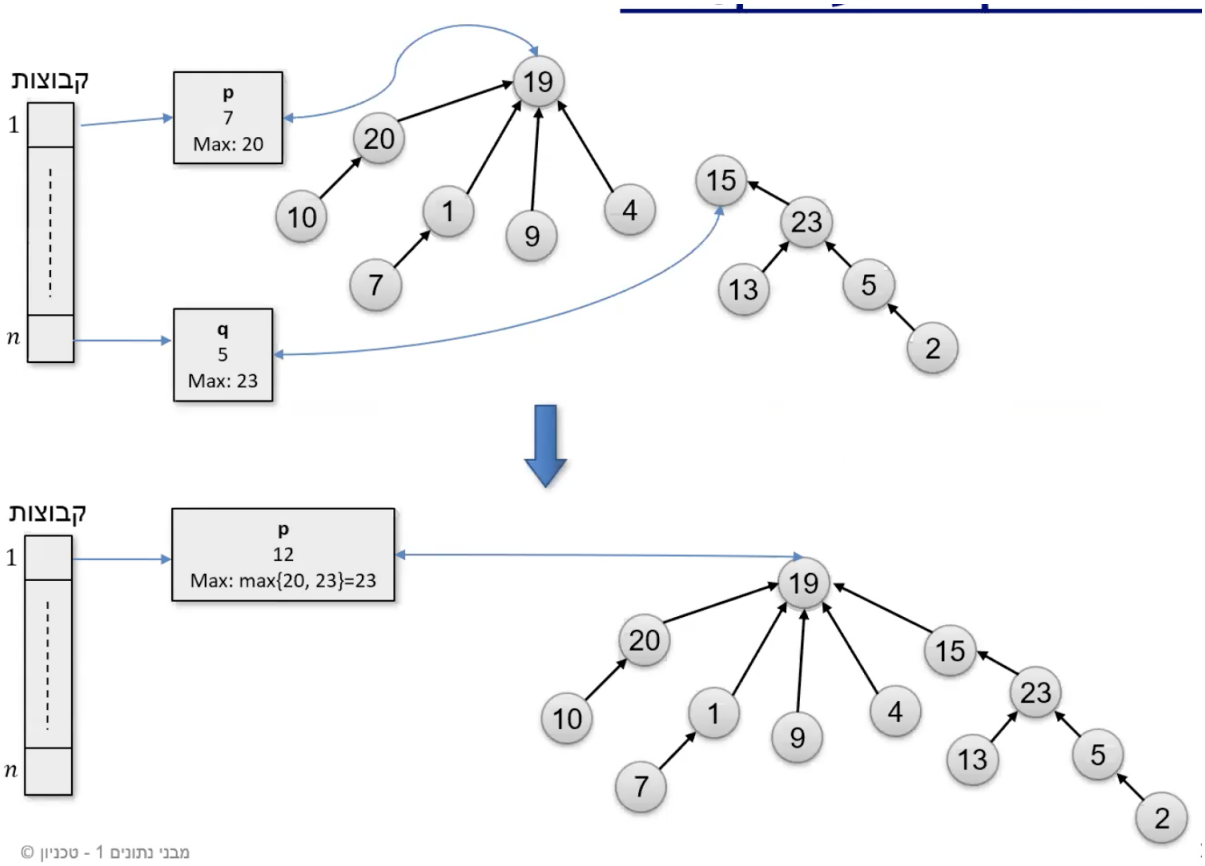
Intiiuition. notice that $log * n$ is close to $O(1)$ at most, but it can reach infinity if we take very very very largne number E.g number of atoms galaxy.

Exercise. (Sets with maximum).
In addition to $Find, Union$ operations we want to add operation $Max(i)$ which will return the maximal element in a set which i exists.

Solution. We will use our data structure $UNnion - Find$ in it's most modified form (compressing pathes and union by size), in a way that our sets describer will hold additional fields max which till us the maximal element in it's set.

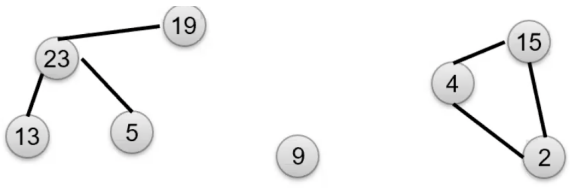
- $Find(i)$ as before.
- $Max(i)$ we will execute $find(i)$ and return the addition field value.
- $Union(p, q)$ We will do union as regularand update tge max field to be the maximal value between max of p, q.

The amortized time complexity of all operations together is log^*n.



Exercise. Given undirected graph with n nodes. In the beginning the graph is empty. Describe a data structure which support the following operations in best time complexity.

- $Connect(x, y)$ - connect x, y nodes.
- $IsReachable(x, y)$ - if we can reach from node x to y.



Connect(23,19)

IsReachable$(19, 13)$	**True**
IsReachable$(19, 2)$	**False**

Solution. We will hold Union-find data structure, the sets in our structure will be connected componenets in our graph. I.e for each element x, the sets in which we will find x there is all elements which are reachable from x.

- $Connect(x, y)$: if $(Find(x)! = Find(y))$ then $Union(Find(x), Find(y))$
- $IsReachable$: $return$ $(Find(x) - = Find(y))$

Exercise. Given a family tree which represented as a inverted directed tree. In the beginning of runnign all the nodes are marked as active. Implement a data structure which suuport the following operations.

- $Close(x)$ - given a pointer to a node x in the tree, mark it us unactive.
- $Master(x)$ - return the most close ancestor of x in the family tree which is marked as active.

Remarks.

- The root can't be closed.
- We can access every element in the family tree within $O(1)$. (E.g element array).

Solution. We will use data structure $Union - find$. We will store a set for each active node by the following rule:

- Each active node x will include a collection if nodes $z$in which $Master(z) = x$.

We will store in the node of each set the name of the active node in the set.

Initalizing: each node is a set itself, since all the nodes are active.

Master(x): We will accesss the tree root in which element x exists and return the name of the single active node in the set.

Close(x): We will denote by y the ancestor of x (we will reach it by our family tree). Then we execute $Union(Find(x), Find(y))$..

Exercise. Given n boxes which are identified by: id, length, height, width.
Rules of building the tower:

(1) Each box has a face which written on it "This Side up" which should be directed up.
(2) A box could be on other box if it's down face obtained in the one above him.

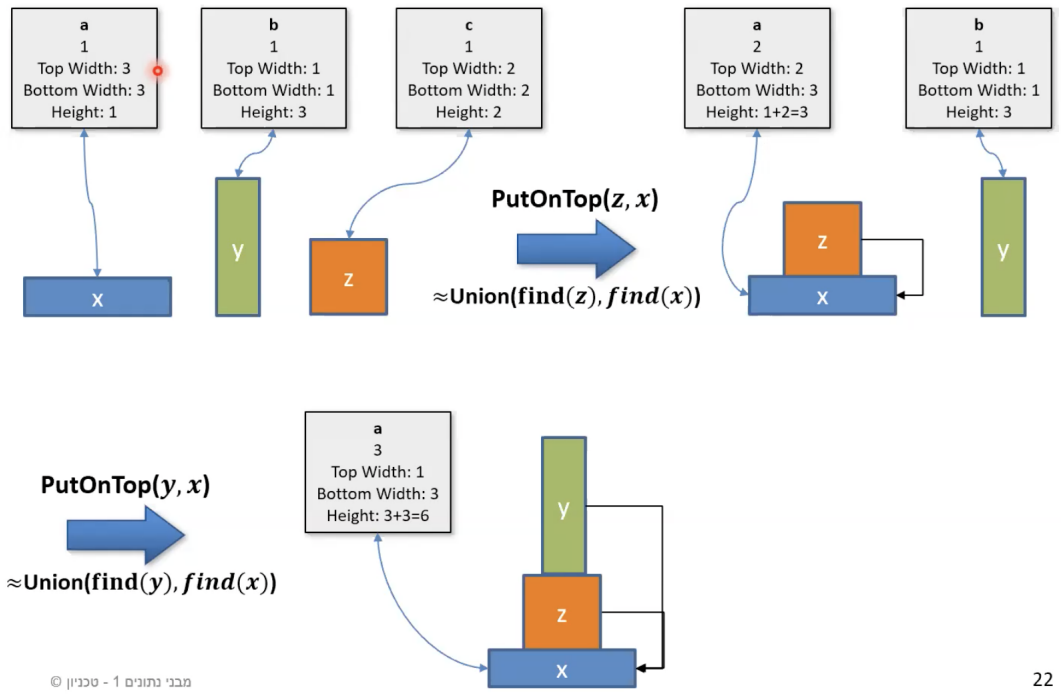
Required a data structure which support the following:

- $Init(Boxes, n)$: Initalizing a storage which is going to store the n containers given in the input Boxes. The containers are identified by the number $1, 2, .., n$ and each container constiture a tower.
- $PutOnTop(B_1, B_2)$: sit the tower in which container B_1 exists on the tower in which B_2exists. (In condition the it satisfy the rules).
- $TowerHeight(B)$: return the height of the tower in which B exists.

Remarks.

- Time complexity of init operation is $O(n)$.
- Amortized time complexity of the two last operations together is $O(log^* n)$,

Solution. We will implement the following using Union-Find. First we look at each box as a set, we will store it's bottom, top width to check if we can sit a box on it. So we do as following;



Now in case we can set a tower on other, we just do union, and update the required.

Formal solution. We will use UF when our set represent a towe. And each element represent a container, each set will store the addition following data:

- The face size of bottom container in the tower.
- The face size of top container in the tower.
- The tower height.

$PutOnTop(B_1, B_2)$:.

- We will do $Find(B_2) \to T_2, Find(B_1) \to T_1$.
- We will insure that the bottom face in T_1obtianed in the top face on T_2.
- if yes, then we execute $Union(T_1, T_2)$ and update the addition data in the new set.

$TowerHeight(B)$ - we dill execute $Find(B)$ and return the height of the tower in which is stored in the additional data of the set.

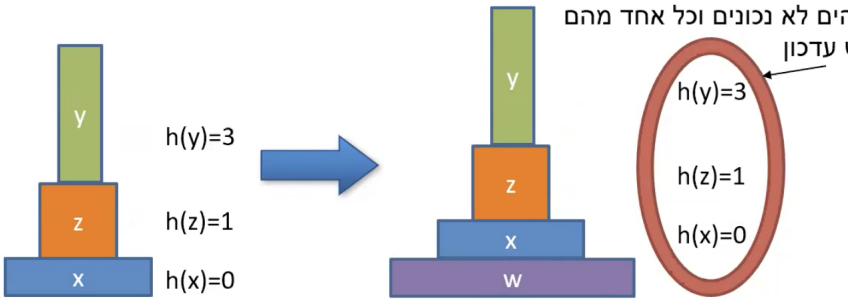
Remark. In each of the following operation we do constant time of $find, union$. hence, the amortized complexity of both is $log^*(n)$.

Exercise. We want to add a function which calculate the height of bottom face of a container from the groud.

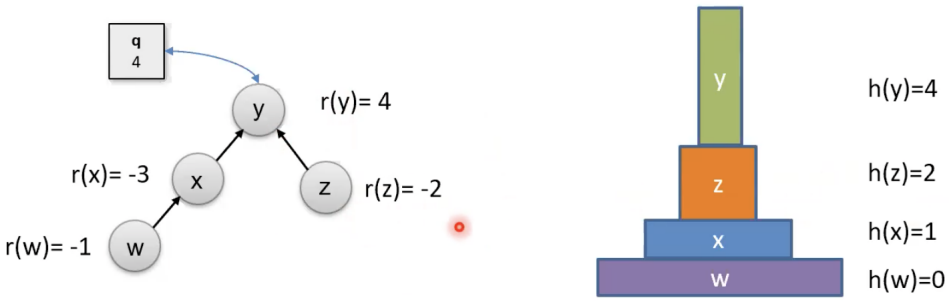
- $HeighFromGround(B)$- return the height of bottom face of container B from ground.

The amortized time complexity of the three operations together is $O(log^*(n))$.

Problem. If we hold a operation using additional data in each node which store the height of container from the ground. Some operations need a update as number of element in the tower which added on the top.



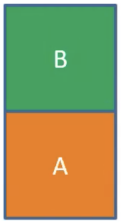
Intituiton for solution. We want to define a additional field for each node in the structure s.t we will do$HeightFromGround$ operation using $Find$, The sum of fields in our $Find$ operation path will be equal to the height of container from the ground.

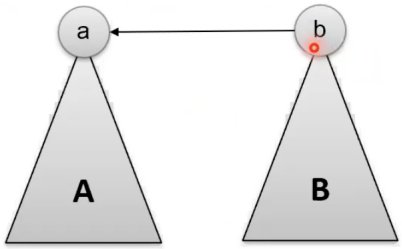


$h(x)$ is the real height of container x from the ground.
$r(x)$ is the addition field which we are going to store for node x in the structure.

Remark. Notice that $h(w) = r(w) + r(x) + r(y) = 0$

Correctness of the property for our structure operation: in order to store additional valje r with the operations PutOnTop. We will assume first that the porperty satisfied for two containers A, B. WLOG $|A| \geq |B|$..





אם הערכים ב-A
שתנו עקב
פעולה?

Notice that the tower is union of two containers A, B. Moreover, the height of boxes in A won't change. Hence, we will define

$$r(a)_{new} = r(a)_{old}, r(b)_{new} = r(b)_{old} + h(A) - r(a)_{new}$$

When a is box in tower A and b is box in tower B.

Remark. Notice that we need to substract $r(a)_{new}$ since in our Find operation we add $r_{new}(a)$ which not needed, so we need to substract it, since each box has a height of it's old height + $h(A)$.

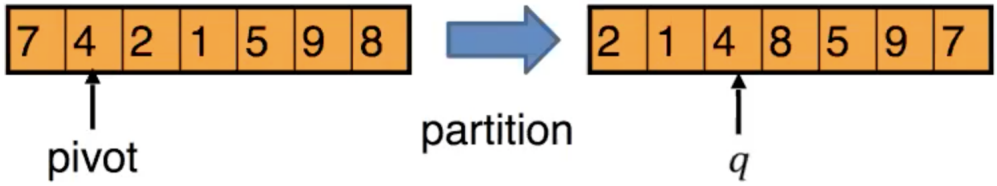
Finding the i element in it size in a array. Can we find the second, third or the i elements in $O(n)$?

We can sortm but each sorting algorithm require $\Omega(nlogn)$. The intiuition to find the i element seem to be hard problem than finding minimal or maxiaml. We will see a solution with $O(n)$ in average. Later we will see how to do it in $O(n)$ at the worst case senario.

Solution. The solution is identical idea of Quichsort. In order to find i element we will do recusrsively the following steps.

- Choose randomly a pivot.
- *Partiotion* : devide the array into to parts. The elements less than pivot are in the left part of the array, and in the right part the elements which bigger than pivot. When pivot is place in the q index.
- if $q == i$ then return pivot.
- if $q > i$ then do recusrively the i element in the left left part of the array.
- if $q < i$ find recusrively the $q - i$ element in the left right part of the array.

Example. Look at the follwing:



Time complexity.

The partiotion operation take $\Theta(n)$ *running time. Better pivot give us less time for the following operation.*

Optimal case. Our pivot is a median. In this case:

$$T(n) = \Theta(n) + T(\frac{n}{2}) \leq c \cdot (n + \frac{n}{2} + \frac{n}{4} + ... + 1) \leq 2c \cdot n$$

This recusrive formula include only one time of $T(\frac{n}{2})$ not as quick sort. Hence, it's solution is linear and not $\Theta(nlogn)$.

Worst case. In each step the array decrease only by 1 *i.e the pivot at the beginning.*

$$T(n) \quad = \quad \Theta(n) + T(n-1) \leq c \cdot (n + (n-1) + ... + 1) = \Theta(n^2)$$

Average case. Our pivot is random i.e.

$$T(n) \leq \frac{1}{n}(\sum_{k=1}^{n} T(max(k-1, n-k)) + O(n)$$

.

המחשה:

$n=6$						
$k=$	1	2	3	4	5	6
$k-1=$	0	1	2	3	4	5
$n-k=$	5	4	3	2	1	0
$Max(k-1,n-k)=$	5	4	3	3	4	5

לכל $\lceil \frac{n}{2} \rceil \leq k \leq n-1$ $T(k)$ מופיע פעמיים.

$n=5$					
$k=$	1	2	3	4	5
$k-1=$	0	1	2	3	4
$n-k=$	4	3	2	1	0
$Max(k-1,n-k)=$	4	3	2	3	4

לכל $\lceil \frac{n}{2} \rceil \leq k \leq n-1$ $T(k)$ מופיע פעמיים,
פרט ל-$T[\lceil n/2 \rceil]$, שמופיע רק פעם אחת.

Since each element appeat twice we get that:

$$T(n) \leq \frac{1}{n}(\sum_{k=\lceil \frac{n}{2} \rceil}^{n} T(k)) + d \cdot n$$

We will see in induction that $T(n) = O(n)$.

Proof. In induction on n. We will show that there is c s.t $T(n) \leq cn$ for all n.

Base. we will take cbig enough S.T $max\{4d, T(1)\} \leq c$..

Step. We assume that $T(k) \leq cl$ *for all* $1 \leq k \leq n-1$ *and we will show that* $T(n) \leq cn$. *Indeed.*

$$T(n) \leq \frac{1}{n}(\sum_{k=\lceil \frac{n}{2} \rceil}^{n} T(k)) + d \cdot n = \frac{2c}{n} \cdot (\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k) + dn$$

$$\leq \frac{2c}{n}(\frac{1}{2}(n-1)n - \frac{1}{2}(\frac{n}{2}-1)\frac{n}{2}) + dn \leq \frac{3cn}{4} + dn \leq_{4d \leq c} cn$$

So we got that $T(n) \leq cn$ for all n by choosing $max\{T(1), 4d\} \leq c$. □

Finding median.

Definition. In a set of n elements, the median is the elements $\lfloor \frac{n+1}{2} \rfloor$ in it's size.

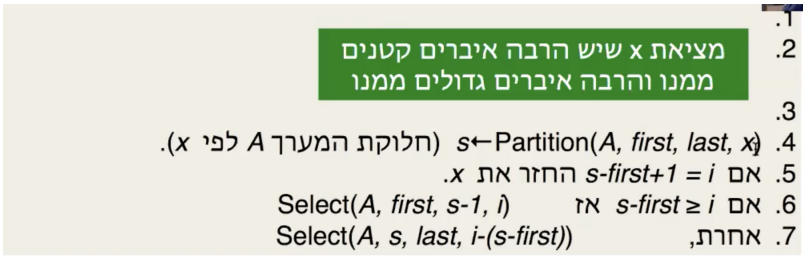
To find the medina we will execute the algorithm to find the $\lfloor \frac{n+1}{2} \rfloor$ in it's size. Time complxity is $O(n)$ in average.

Find the i element in it's size in a determanstic way.

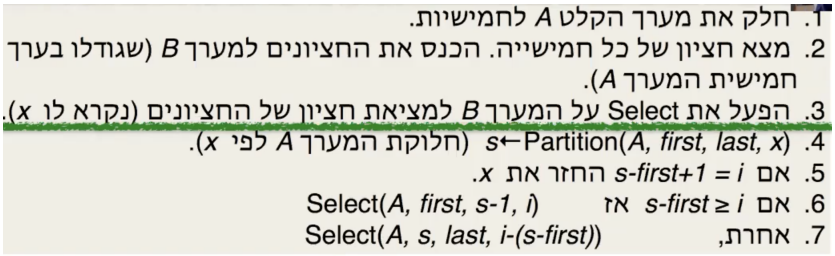
- The time required for the algorithm we descirbed rely on the array size in each recusive call.
- Target: To insure that in each recusrive call the array is decreased properly, in order to insure wunning time $O(n)$ in worst case.
- We will make algorithm in each step will do a partiotion to a two small sets than the source set.

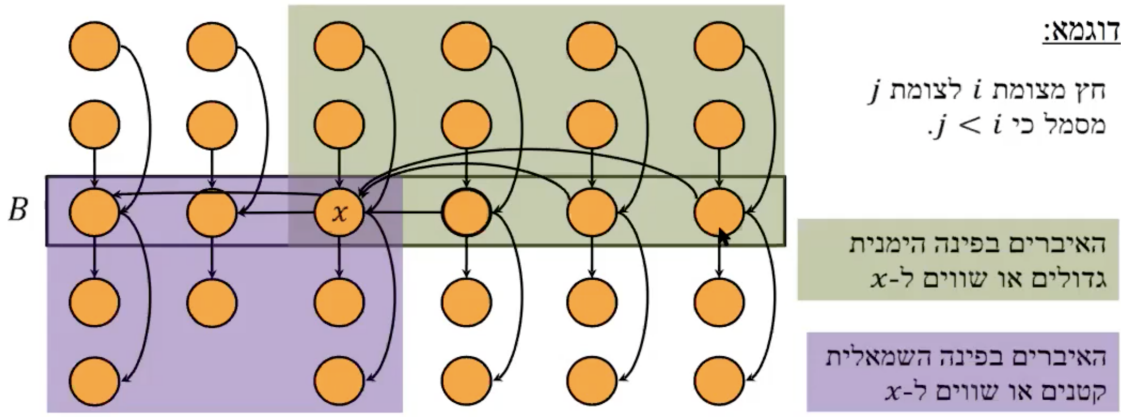
- Each set size is less than α times the source of the source set when $0 < \alpha < 1$.
  - Notice that α is a constant which not depend on the set size.

Select algorithm. This algorithm will let us find element with i positoion within $O(logn)$ at the worst case. We do that by each recusive call we choose a good pivod in which many element less than it and many elements bigger than it. Then we continue with the algorithm we saw beforem but now we save a lot of time. So the algorithm will be as following.



We can see lat how are we going to find this good pivot.





What are we doing?

first we devide our array into $\frac{n}{5}$ subarrays which in each we can find the median. Then we take the median of the $\frac{n}{5}$ median elements i.e median of the median. In the following way we can insure that our pivot is bigger than $\frac{n}{5\cdot 2} \cdot 3$ element. And less than $\frac{n}{5\cdot 2} \cdot 3$ elements.

Lemma. *Number of element in select recusrsive calls is at most* $\frac{7n}{10} + 3$.

Proof. Since x is median of median and tere is $\lceil \frac{n}{5} \rceil$ median, implies that at least $\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil$ medians less equal than x.

- In each set of small median less equal than x there is 3 element which are less/equal than x (expect one set which has no 5 elements) Hence, number of element which are less equal than x is at least:

$$(\lceil \frac{1}{2} \cdot \lceil \frac{n}{5} \rceil - 1) \cdot 3 \geq \frac{3}{10}n - 3$$

- In identiacal way the number of elements which are bigger equal than x is at least:

$$(\lceil \frac{1}{2} \cdot \lceil \frac{n}{5} \rceil - 1) \cdot 3 \geq \frac{3}{10}n - 3$$

- So from here in each recusrive calls we have at most $\frac{7n}{10} + 3$ elements.

□

Corollary. *The recursive formu;a of time running is*

$$T(n) \leq T(\lceil \frac{n}{5} \rceil) + T(\frac{7}{10}n + 3) + O(n)$$

Lemma. $T(n) = O(n)$.

Proof. In induction on n. We will show that there is c in which $T(n) \leq cn$ for all n.

Base: let c constant in which for all $n \leq 80$ we have $T(n) \leq c \cdot n$ for all n..

Induction assumption + step: We assume that for all $k < n$ satisfied that $T(k) \leq c \cdot k$ and we will show from here that stem $T(n) \leq c \cdot n$. Let d be the constant hidden in the O mark.

$$T(n) \leq c \cdot \lceil \frac{n}{5} \rceil + c(\frac{7}{10} + 3) + d \cdot n \leq$$

$$c \cdot \frac{n}{5} + c + c \cdot (\frac{7n}{10} + 3) + d \cdot n$$

$$\leq c \cdot \frac{9}{10}n + 7c + d \cdot n \leq cn$$

When $c > 80d$ the last inequality satisfied for a;; $n \geq 80$:

$$\cdot \frac{9}{10}n + 7c + d \quad < \quad \frac{9cn}{10} + 7c + \frac{c}{80}n = \frac{73}{80}cn + 7c$$

$$= cn + (-7c\frac{n}{80} + 7c) \leq cn$$

□

We we devide by 5?

נוסחת הנסיגה עבור זמן הריצה מקיימת:

עבור חמישיות: $T(n) \leq T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7}{10}n + 3\right) + O(n)$

עבור שלשות: ~~$T(n) \leq T\left(\left\lceil \frac{n}{3} \right\rceil\right) + T\left(\frac{2}{3}n + 2\right) + O(n)$~~

עבור שביעיות: $T(n) \leq T\left(\left\lceil \frac{n}{7} \right\rceil\right) + T\left(\frac{5}{7}n + 4\right) + O(n)$

We notice that $\frac{n}{5} + \frac{7}{10}n \leq n$ so it work while in deviding by 3 it won't.

Exercise. Can we insure that *Quicksort* run with $O(nlogn)$ at the worst case?

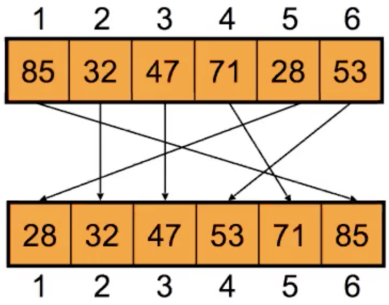
- YES.
- In each step we find the pivot with $O(n)$ and we did partiotion with $O(n)$ and we call quicksort got both subarrays we got thereofore, the time complexity is:

$$T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$$
$$T(n) = \Theta(nlogn)$$

Lower bounder for comparing sorting algorithms.

- Given two keys k_1, k_2 can checl $k_1 \geq k_2 \vee k_2 \geq k_1, k_1 > k_2, k_1 < k_2$.
- Not by comparing: Is k_1 even? negative? what is the third bit in tbinary represntation of k_1? what is the difference between k_1 to k_2.
- All the algorithm we saw till now are baseed on comparing.
  - Bubble sort, merge sort, heap sort, quick sort.
- Advantage in comparing:
  - We can sort each type, without addition condintion.

Sorting as finding permutation. Assumin we get a array with n different elements and we want to order them sorted. Each sequence of n different numbers define a permutation π on indexes E.g :



$\pi(1) = 6, \pi(2) = 2, \pi(3) = 3,$
$\pi(4) = 5, \pi(5) = 1, \pi(6) = 4$

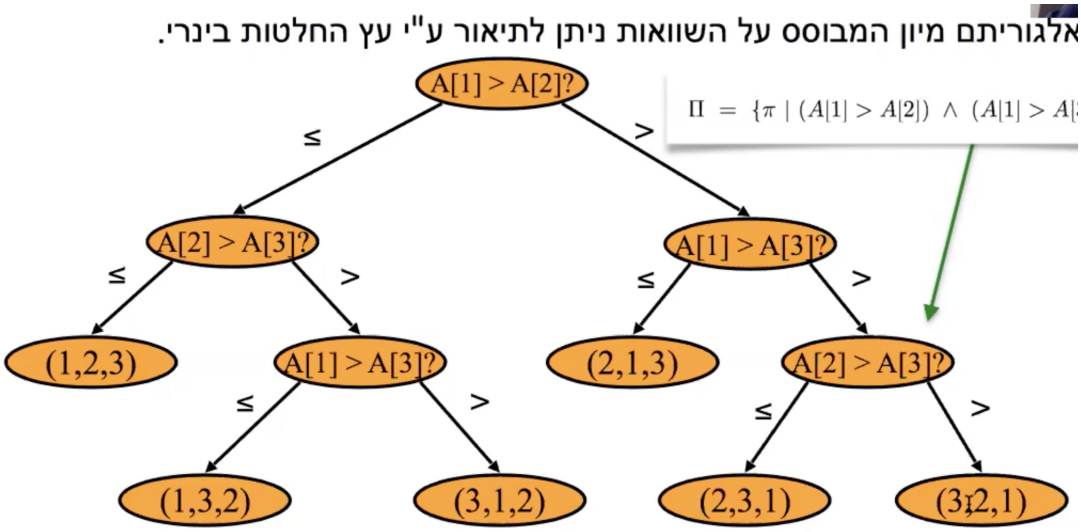
וכן פרמוטציה הופכית $\sigma=\pi^{-1}$:

$\sigma(1) = 5, \sigma(2) = 2, \sigma(3) = 3,$
$\sigma(4) = 6, \sigma(5) = 4, \sigma(6) = 1$

A algorithm get as input array A and find as output a permutation σ s.t

$$A[\sigma(1)] < A[\sigma(2)] < A[\sigma(3)] < ... < A[\sigma(n)]$$

Each permutaion is possible since the element $A[i]$ can be mapped to any position in the output array. There is $n!$ permutation. How many comparing operation required in order to determine a permutation. We can illustate that by a decision tree.



\

Remark. Notice that each path in the following binary tree constitute a other permutation so we can't get two identical permuation at two different leaves in the tree. Since there is $n!$ leaves then out tree height id at least $log(n!)$ and this is number of compares required to do. We sow already that $log(n!) = \Theta(nlogn)$.

Corollary. *Each comparing sorting algorithm is bounded by* $\Theta(nlgon)$ *complexity.*

Bucket sort.

Target. Sorting n *different number in range* $1, ..., k$.

Method. We will use bboolean array in length of k..

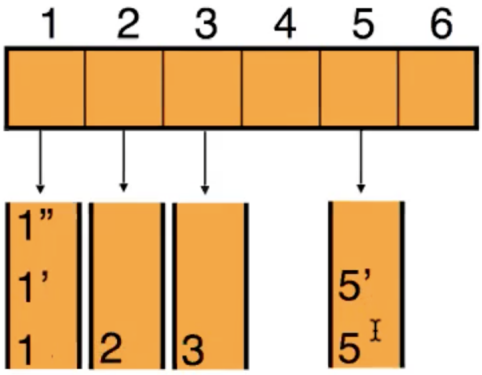
```
                                        .A אפס את   .1
A[x] = 1                                :x לכל בקלט בצע   .2
for (i = 1; i < k ; i++) {              עבור על המערך ואסוף   .3
        if (A[i] = = 1 ) output i ;
}
```

Example. Assuming our range is $k = 1..6$ for input $5, 1, 3, 2$.

Time complexity. Initlaizing the array with k *element to* 0. *Walk on the array* $O(n)$. *Get all* k *values* - $O(n+k)$. If the range size is $k = O(n)$ then we get algorithm in linear time $\Theta(n)$.

Target: Sorting n *element which are neccessary different from ange* $1, .., k$..

Method. Assuming our range is $1, .., k = 6$ *and our input is* $5, 1, 3, 2, 1, 1, 5$..

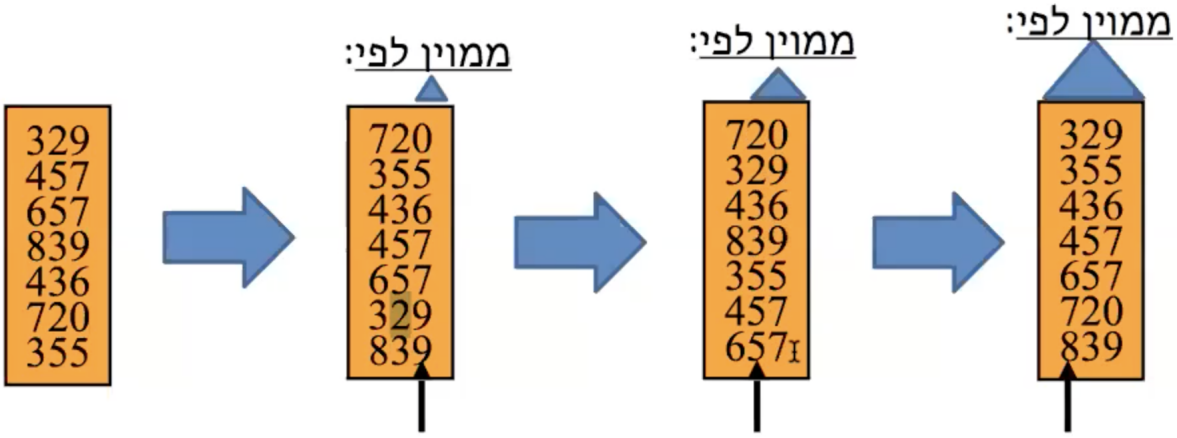


Definition. A sorting method is called stapbe is the following condition satisfied, if in the input $A[i] = A[j]$ and $i < j$ then $A[i]$ we appear before $A[j]$ in the output.

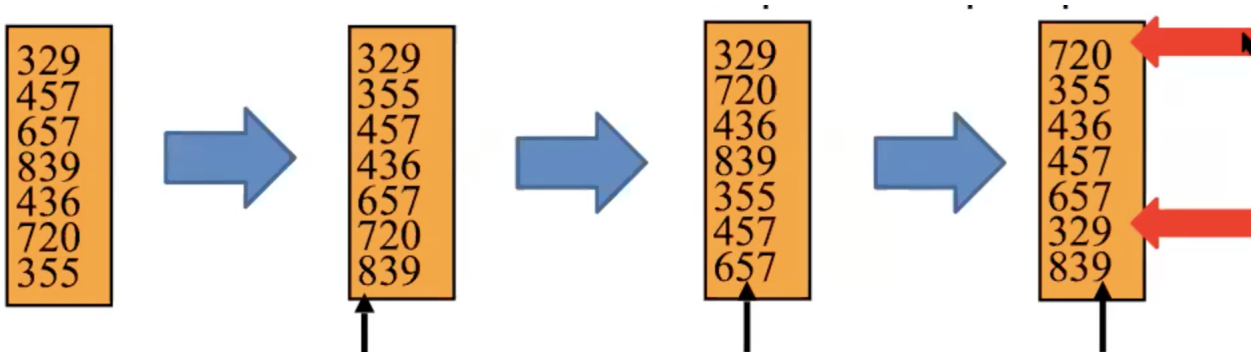
Corollary. *Stem that Bucketsort is a stable method.*

Observation. Bucketsort method is not effcient especially when the n number are taken form a big range $1, , .., k$ E.g $n << k$. For instance sorting $1, 1003, 9999$ will require big array 10^5 places when $n = 3$ (as the case we saw in the hash table).

Solution. We will assume that each key of d digits, We will sortnn each digit seprately using the stable sorting method (E.g bucketsort). The LSB bits are sorted first.



Notice that we can' sort from MSBA TO LSB for the following reason.



Time complexity analysis. For each digit we will do Bucketsort within time of $\Theta(n+10)$ for d digits that time required is $\Theta(d(n+10))$

E.g for number is base 10 *with d number of digits (which not depend on the number* n*), the sorting is executed in* $\Theta(n)$*..*

In general. for numbers presented in base r*, for each digit we will do bucketsort in time of* $\Theta(n+r)$ *for d digits, the general time is* $\Theta(d(n+r))$*..* When our maxima; element is k, number of digits in base r is $log_r k$ then the time required is $\Theta(log_r k \cdot (n+r))$.

E.g for $k = O(n^c)$ *for c constant and base r, the timre required is* $\Omega(nlogn)$*. As required in sorting with comparing only.*

Heap exercises.

Minimum Heaps.

Definition. A data structure which is defined as following:

- $MakeHeap$ - Constructing a heap with n input elements.
- $Insert(x)$ - Inserting element x to the heap.
- $DecKey(p, x)$ - Decreasing the of node p to x.
- $FindMin$ - Finding the minimal element in the heap.

- $DelMin$ - Removing the minimal element from the heap.

Uses. Priority queue.E.g in medical clinic when our queue is pations.

Implementing by AVL tree.

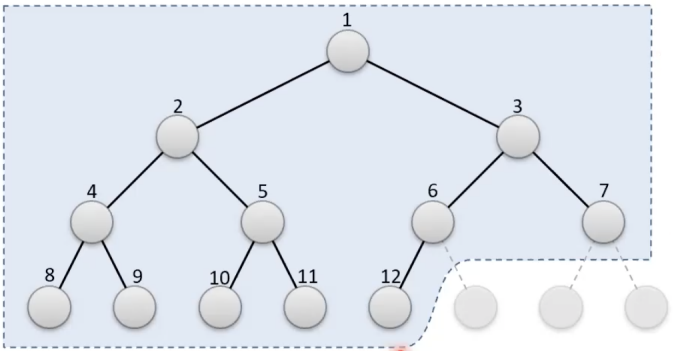
- $MakeHeap$ - Inserting each element to the tree. $O(nlogn)$.
- $Insert(x)$ - Inserting element x to the tree and updating the minimum $O(logn)$.
- $DecKey(p, x)$ - Deleting node p and inserting it again to the tree with value x.
- $FindMin$ - Returing the value of the node pointed as minimum.
- $DelMin$ - Removing the minimum from the tree and find new minimum.

Remark. Implementing the heap using a almost complete trr which will improve the complexity of $MakeHeap$ to $O(n)$.

Almost complete tree.

Full tree. Binary tree in which for each interior node there is exactly two childs and all the leaves are at the same level tree.

Almost complete tree. full tree in which we omit from the lower level. Starting from the most right node.

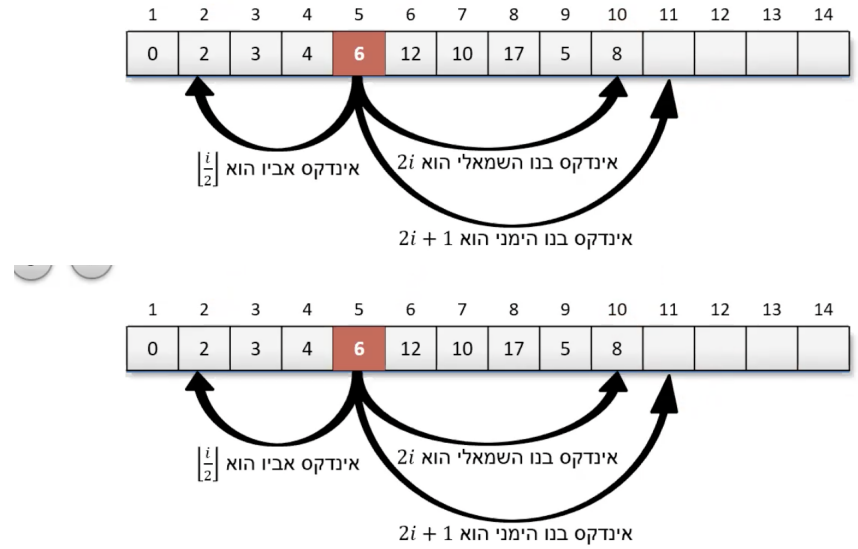


We will number the node by the lines order. From left to right. We will call those numbers indicents.

Properties of almost complete tree.

(1) Almost complete tree with n nodes and height of h satisfy: $2^h \leq n \leq 2^{h+1} - 1$. From here $h = O(logn)$.
(2) Number of leaves is $\lceil \frac{n}{2} \rceil$.
(3) Number of interior nodes is $\lfloor \frac{n}{2} \rfloor$.
(4) The index of the root is 1. The last leave has index of n. A new node which will be inserted will get index of $n + 1$.
(5) For interior node i :
    - The index of it's left child is $2i$. And it's right child is $2i + 1$.
    - The index of it's father of $\lfloor \frac{i}{2} \rfloor$.
(6) Node is a leaf if $2i > n$.

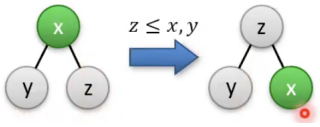
Implementing a heap as a almost complete tree. We will use a array when a bounder given on number of elements in the heap. Using the properties described. We can set a almost complete tree in array entries.



Heap rule (minimum rule). Each child bigger than it's father. Therefore, the minimum on heap will be in the root.

Sift Down.

Algorithm. If x bigger than it one childs, then change x with it's minimal child.



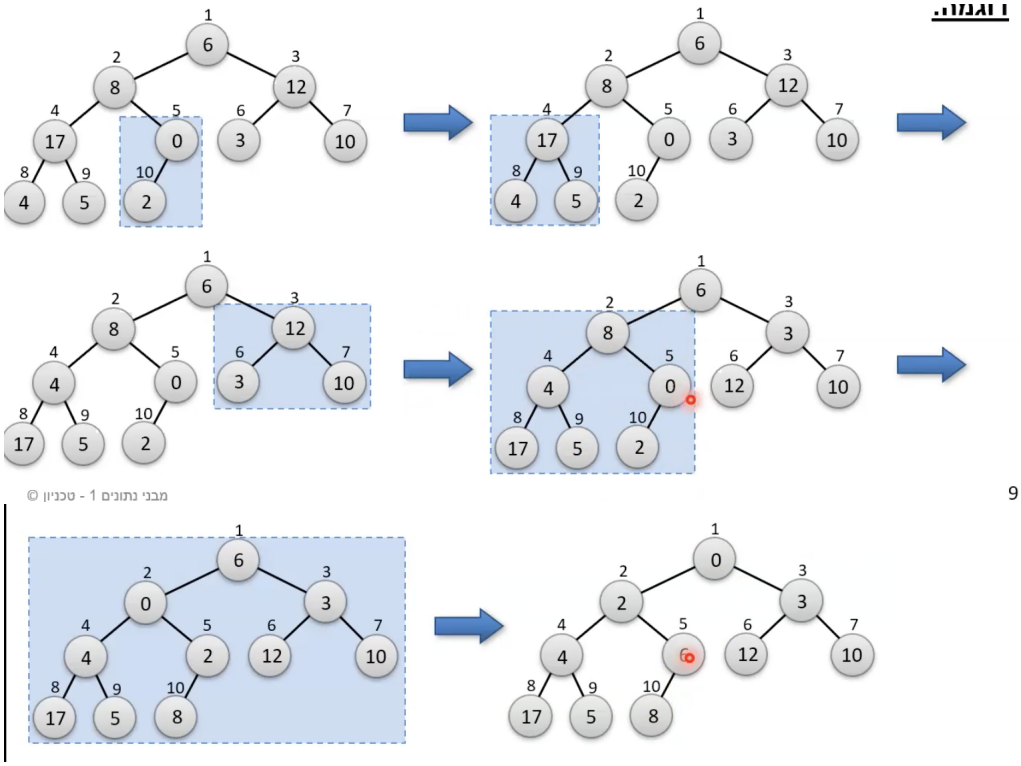
נכונות:

Complexity: $O(h)$ when h is the input height.

Make heap. Given n elements, the following operation build a minimum heap which contain the following elements in complexity of $O(n)$. For that, we are going to define $SiftDown$. We will use $SiftDown$ operation as the following, in order to implement $MakeHeap$:

Algorithm: for all i from $\lfloor \frac{n}{2} \rfloor$ to 1. We execute $SiftDown$ on the node i in the tree.

Example. As the following:



Complexity.

$$T(n) \leq \frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + ... + 1 \cdot log(n)$$

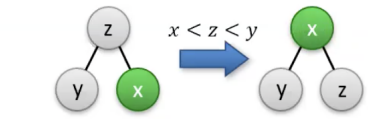
$$\leq n \cdot [\frac{1}{4} + \frac{2}{8} + \frac{3}{16} + ...] \leq$$

$$n \cdot \begin{bmatrix} (\frac{1}{4} + \frac{1}{8} + \frac{1}{16} + ...) \\ (\frac{1}{8} + \frac{1}{16} + ...) \\ (\frac{1}{16}...) + ... \end{bmatrix} \leq \quad n \cdot [\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ...] = n = O(n)$$

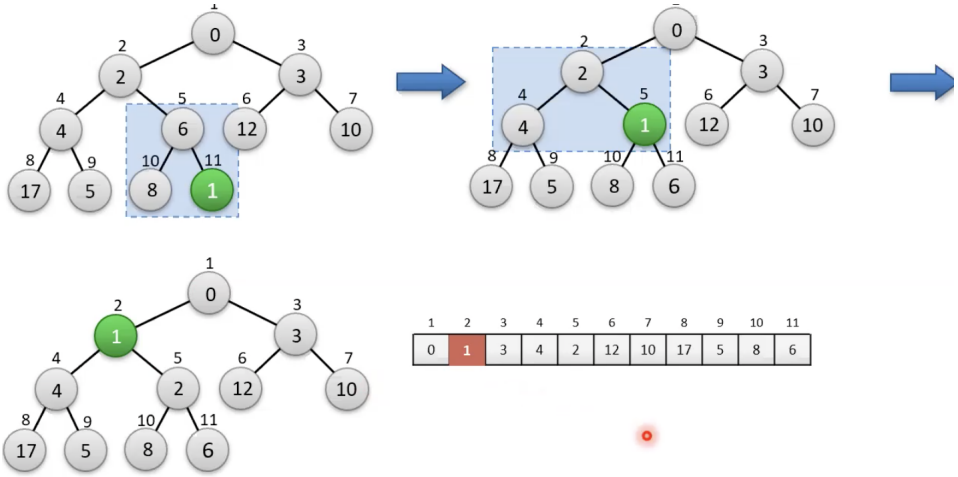
$Insert(x)$: we will add a new element x as new leave. (In the first free index). In order tofix the heap, we will define a helping oepration $SiftUp$..

Sift Up:

Algorithm. Each time x is less than it fathers. Change between x and it's fathers.



Complexity. $O(h)$ when h is the input height.



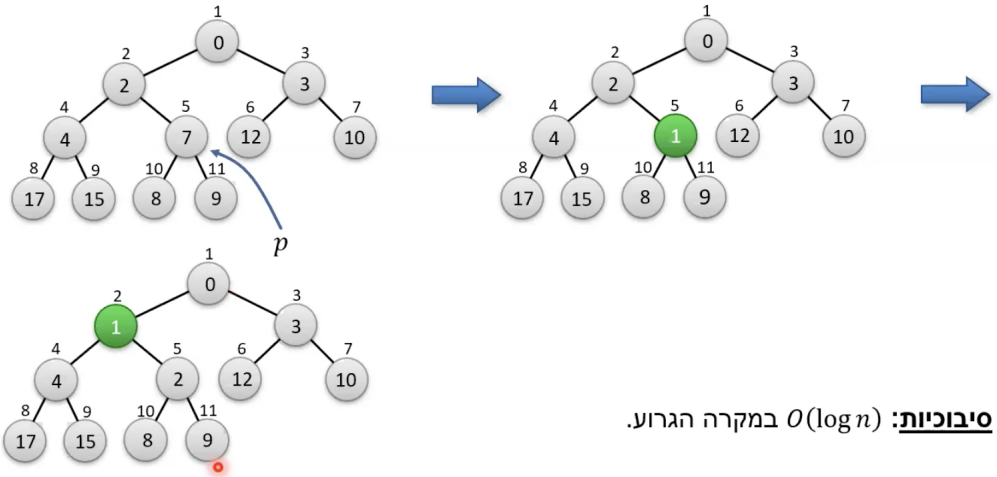
Example.

Complexity. The insertion operation, together with Sift Up, costs $O(logn)$ at the worst case.

Dec-Key(p,x) - decreasing a key.

Algoeithm: if the value in p less than x then do nothing. Otherwise, change the value of p to x and execute SiftUp to the node pointed to.

DecKey(*p*,1) **דוגמה:**



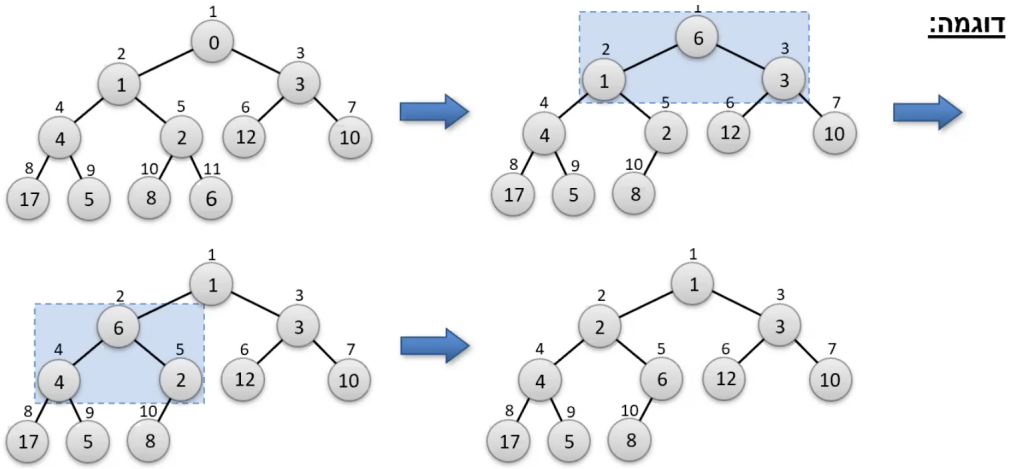
סיבוכיות: $O(\log n)$ במקרה הגרוע.

Example.

Complexity. $O(logn)$ *in worst case.*

Del-Min . We will change between the root and the last leave then we will delete the minimal element. (Which appear before as root. Then we do Sift Down on thw new root in order to get valid heap.

דוגמה:



סיבוכיות: $O(\log n)$ במקרה הגרוע.

If there is no bounder of number of element. We can implement also a minimum heap as almost complete tree without array.

The change we need to do for each of operations,

Make Heap:

- We will walk on interior nodes in postorder tranversal.
- The operation Sift Down require the two pointer to child. Point to a valid heap.

DecKey: We do Sift up from the node,

Insert. We will add leave and exxecute sift up from him.

Del min: We will change between the last leave and root. Then sift Down from the root.

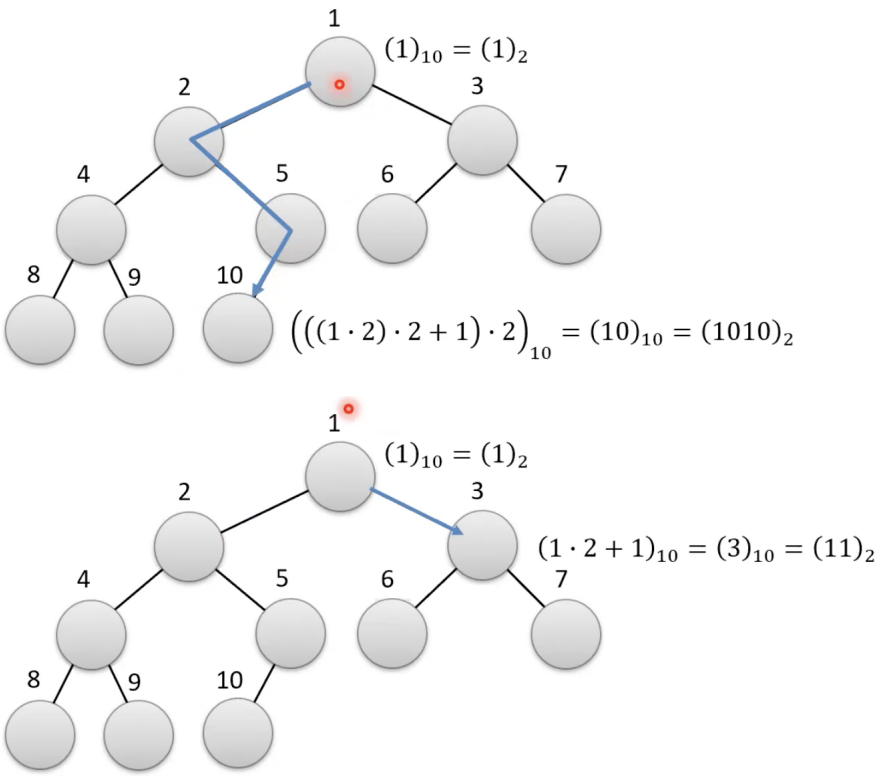
> How are we going to find the last leave with index i in $O(logn)$ at worst case?

Solution. We remember the properties of almost complete tree.

- The root has index 1 always.
- For node i it's left child index is $2i$. The index of it's right child is $2i+1$.

We can think about going to right as 1 and going to left as 0 and the binary number is actually i as in the previous represnation but represented in decimal.

Example. Look ath the following node with index i.



Corollary. *To find a node with index i :*

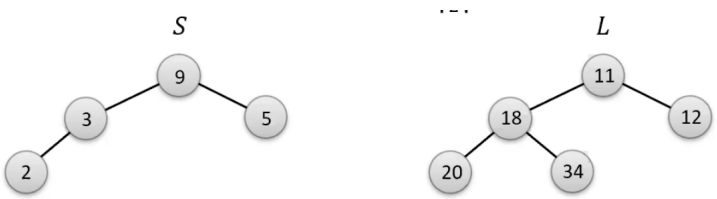
(1) We look at it's binary representation of i,
(2) For each bit, starting from the MSB bit, We go left child if 0 and right child if 1 starting from the tree root.

Exercise. Given a linked list of numbers and also given the median of the following list. Consider the median as the smallest number in the structure which is bigger than $\lfloor \frac{n}{2} \rfloor$ elements in the structure. Implement a data structure which support the following operations.

- $Init$: Init the structure in complexity of $O(m)$, when m is number of elements in the list.
- $Insert(x)$: Add x to the structure with complexity of $O(logn)$, when n number of elements in the structure.
- $FindMed$: return the median value in $O(1)$.
- $DelMed$: Delete the median from the structure in $O(logn)$.

Solution. We will hold two heaps:

- Maximum heap S which contain all $\lfloor \frac{n}{2} \rfloor$ smallest elements.
- Minimum heap L which will contain $\lfloor \frac{n}{2} \rfloor$ biggest elements.



Remark. We notice that we have to insure $|L| = |S|$ or $|L| = |S|+1$ by the structure definition.

Init : we will split the input element to two sets. The first is smaller than the median and the second bigger than the median elements. Now, we will execute make heap on one of each of the sets in order to get two heaps. So in total we have:

$$O(m) + 2 \cdot O(\frac{m}{2}) = O(m)$$

FindMid : The median is the root of L so we return it by $O(1)$.. *Insert*(x): if x is less than the median then we insert it to S. Otherwise to L. In case a fix required we fix the order of the heaps L, S.

- If $|L| = |S| - 1$ then we move the root of heap S to L.
- If $|L| = |S| + 2$ then we move the root of heap L to S.

DelMed: we delete the root of L if then $|L| = |S| - 1$ *then we move the root of heap S to L..*

Complexity: The complexity of the latest opearions take $O(3 \cdot log(\frac{n}{2})) = O(logn)$ *at the worst case.*