```
/** CLASSE PARTICIÃM-^S **/
/*
    Aquest mÃ²dul tÃ© com a objectiu emmagatzemar una colÂ•lecciÃ³ d'elements de
 tipus T,
    classificats en diferents grups. ÃM-^Is a dir, mitjanÃ§ant una AVL, anem gua
rdant els
    diferents elements en grups diferents amb possibilitat d'uniÃ³, on cada grup
 tÃ© el
    seu propi representant.

    Hem decidit utilitzar l'algoritme d'emmagatzematge AVL en comptes de per exe
mple Taules Hash.
    AixÃ² Ã©s degut al fet que tot i tenir una complexitat de O(1) que supera la
 de l'AVL que s'espera que
    sigui O(log n), ens convÃ© mÃ©s a la llarga tenir les dades ordenades com a
l'AVL (ja que per
    exemple a la classe dedalus guardarem posicions a particiÃ³ i Ã©s preferible
 que estiguin en ordre)
    que no pas el millor cost de les Taules Hash. A mÃ©s, pel que fa a memÃ²ria
una AVL Ã©s mÃ©s eficient, perquÃ¨ no
    reserven mÃ©s memÃ²ria de la que els cal. TambÃ© cal tenir en compte que est
em treballant amb
    dades dinÃ miques i que anem modificant la mida del nombre total d'elements,
 per tant, en fer insercions
    de nous elements ,l'AVL torna a ser una gran opciÃ³ pel que hem comentat ant
eriorment.

*/

/** ATRIBUTS PRIVAT **/
/*

    Per a la realitzaciÃ³ d'aquesta classe hem hagut de crear diversos atributs
privats.

    Per comenÃ§ar hem creat tres variables del tipus nat, el primer Ã©s per a la
 identificaciÃ³ del nombre mÃ xim de grups, el
    segon pel nombre de nodes i el tercer pel nombre d'elements de l'AVL.
    Aquestes variables ens seran Ãºtils per consultar si s'han unit grups, per v
eure la quantitat d'elements que tenim a l'AVL
    i per saber el mÃ xim de nodes possibles que podem tenir a la particiÃ³, res
pectivament.

    Seguidament, hem trobat necessari crear un struct anomenat node, per a la re
presentaciÃ³ de l'AVL. Dins d'aquest struct
    anirÃ  la informaciÃ³ de la clau d'un node, un punter destinat a mirar que h
i ha a l'esquerra de l'AVL des del node actual i
    un altre per mirar a la dreta. TambÃ© una variable del tipus nat per tenir u
n seguiment de l'altura mÃ xima on es troba el node,
    i un altre per saber, d'aquest node, quants fills en tÃ©. Finalment, tambÃ©
necessitem un punter representant que apunti al pare del
    node actual, aquest representant serÃ  important per a la identificaciÃ³ de
grups.

    Ja per acabar, l'Ãºltim atribut privat Ã©s un punter que apunta a un node. U
tilitzant aquest punter podrem accedir a tota la
    informaciÃ³ emmagatzemada al struct.


*/

nat max_grup;   //nÃºmero mÃ xim de GRUPS
```



```
nat n_max;  //nÃºmero mÃ xim de NODES
nat n_elem;       // Nombre de nodes del subarbre
struct node {
  T _k;      // Clau
  node* _esq;   // fill esquerre
  node* _dret;  // fill dret
  nat alt_max;
  node* representant;
  nat fills;
};
node *_arrel;

/** MÃM-^HTODES PRIVATS **/
/* Pel correcte funcionament de la classe particiÃ³, ha estat necessari fer Ãºs
de mÃ¨todes auxiliars, a continuaciÃ³ un breu resum de cadascuna a part
del PRE i POST del .cpp*/

//MÃ¨todes Constructora,Destructor i CÃ²pia

node* copia_particio(node *n);
/* Aquest mÃ¨tode auxiliar fa la implementaciÃ³ de copiar la particiÃ³ que ens
donen a partir del seu node inicial. Ha estat necessÃ ria per poder fer
recursivitat per recÃ³rrer tant la nostra particiÃ³ com la passada per p.i*/

void destrueix_particio(node* p);
/* Aquest mÃ¨tode auxiliar fa la implementaciÃ³ de destruir la particiÃ³ que ens
donen a partir del node inicial passat per parÃ metre.
Ha estat necessÃ ria per poder fer recursivitat.*/

void copia_repre(node* p, node* a, node* auxiliar);
/*Aquest mÃ¨tode auxiliar fa la implementaciÃ³ recursiva de la cerca
del pare o representant del node p, passat per parÃ metre, per desprÃ©s poder-ho
 aplicar
a la nostra particiÃ³.
ÃM-^Is a dir, perquÃ¨ la cÃ²pia sigui exacta, tambÃ© ha de tenir en compte els r
epresentants,
per aixÃ² fa un recorregut simultani en ambdÃ³s particions assignant els represe
ntants.
Sense aquesta funciÃ³, desprÃ©s de fer una cÃ²pia no funcionaria de forma
correcta la uniÃ³*/

///MÃ¨todes d'afegir

node* insereix_avl(node *n, const T &k);
/* Aquest mÃ¨tode auxiliar Ã©s qui fa tota la funciÃ³ d'afegir. Ha estat necessa
ri
fer aquesta funciÃ³, ja que requereix fer recursivitat. Dins d'aquest mÃ¨tode ta
mbÃ©
es fan crides a altres funcions auxiliars*/

node* newNode(T k);
/*Aquest mÃ¨tode auxiliar tÃ© com a funcionalitat crear nous nodes inicialitzats
amb els atributs privats. Fem aquesta funciÃ³ com a alternativa a la funciÃ³
inicialitzadora que hi ha per exemple en la classe diccionari treballada
a classe.*/

static nat altura_max(node *n);
/* Aquest mÃ¨tode auxiliar tÃ© com a objectiu retornar l'altura mÃ xima del node
passat per parÃ metre en cas que no estigui buit. S'ha fet en una funciÃ³ auxili
ar
per comoditat, ja que en cridar-la ja saps que tÃ© en compte el cas on el node
ja estigui buit. */
```

```
static nat max(nat a, nat b);
/*Aquest mÃ¨tode auxiliar Ã©s necessari per a la comparaciÃ³ d'elements de
forma rÃ pida i eficient. Aquesta funciÃ³ es crida diverses vegades en
altres mÃ¨todes auxiliars*/

static nat factor_eq(node *n);
/*Aquest mÃ¨tode auxiliar Ã©s utilitzat per comprovar si els dos subarbres de l'
AVL
estan o no equilibrats. Si retorna 0 significa que sÃ- que ho estÃ , en cas
contrari no ho estarÃ */

static node* rightRotate(node *y);
/*Aquest mÃ¨tode auxiliar fa que donat un node de l'AVL sigui reordenat perquÃ¨
aquest estigui equilibrat. En aquest cas la rotaciÃ³ serÃ  cap a la dreta.
Una vegada rotat, s'actualitzen les altures de l'arbre.
*/
static node* leftRotate(node *x);
/*Aquest mÃ¨tode auxiliar fa que donat un node de l'AVL sigui reordenat perquÃ¨
aquest estigui equilibrat. En aquest cas la rotaciÃ³ serÃ  cap a l'esquerra.
Una vegada rotat, s'actualitzen les altures de l'arbre.
*/

//MÃ¨tode find

node* find(node *n) const throw(error);
/*Aquest mÃ¨tode auxiliar recorre tot l'AVL buscant el representant del node
passat per parÃ metre. Aquest mÃ¨tode Ã©s molt Ãºtil per exemple a la funciÃ³
unir, ja que requereix saber qui Ã©s el representant d'un grup per a unir-los.*/

//MÃ¨todes d'UNIR

bool existeix(node *n, T e, bool trobat) const throw();
/*Aquest mÃ¨tode auxiliar, de forma recursiva, va cercant si l'element passat pe
r parÃ metre
existeix dins de la particiÃ³ o no. Tenint aquesta funciÃ³, desprÃ©s Ã©s mÃ©s fÃ
 cil
gestionar altres mÃ¨todes, doncs, si no existeix l'element, ja no es fa res o es
 treu un error.*/

node* buscanode(node *n, T e) const throw();
/*Aquest mÃ¨tode auxiliar Ã©s un tÃ-pic recorregut de cerca d'una AVL de forma i
terativa.
Serveix com ajuda per trobar el punter node a l'AVL de l'element passat per parÃ
 metre.*/
```