Persistència i àmbit de les variables

```
int f() {
  int x;  /* no te relacio amb x de g(); */
  ...
}
int g() {
  double x;  /* no te relacio amb x de f(); */
  ...
}
```

Les variables “locals” només són visibles dins de la funció on s’han declarat; i existeixen mentre dura l’execució de la mateixa, desapareixent de la memòria en acabar (**automàtiques**).

Persistència i àmbit de les variables

En llenguatge C es poden declarar variables **fora** de les funcions: s'anomenen variables **globals**, a més, són permanentment en memòria durant l'execució del programa (**estàtiques**).

Les variables globals són visibles en totes les funcions del fitxer definides amb posterioritat a la seva declaració.

Si es vol usar una variable global definida en un altre fitxer de codi, en els fitxers on no es declarada inicialment cal posar **`extern`**

```
extern int a; /* var. global d'altre fitxer */
```

Persistència i àmbit de les variables

```
int a;
/* a es visible en main(), f() i g(), no en h() */
int main(void) { ....  }
int b;
/* b visible en f(),g() i h(), no en main() */
int f() {
        int x;  /* no es x de g() */
        ...  }
int g() {
        int x;  /* no es x de f() */
        ...  }
int h() {
        double a;  /* amaga la variable global a */
        ...  }
```

Cerca en vectors ordenats

Considerem $v = (v_0, v_1, ..., v_{n-1})$ tal que

$$v_0 \leq v_1 \leq \cdots \leq v_{n-1}$$

Problema: Donat x, $\exists i \in 0, \ldots, n-1$ amb $v_i = x$?

Nota: si s'han de trobar **TOTS** els i que ho compleixen, caldria retocar els algorismes.

Cerca lineal

```
for (j=0; j<n && v[j]< x; ++j);

if ( j == n || v[j] > x ) {
   /* no hi es */
} else {
   /* hi es, v[j] == x */
}
```

Calen $O(n)$ comparacions per a trobar l'element

Cerca binària

```
esq = j = trobat = 0; drt = n-1;
while (esp <= drt) {
  j = (esq+drt)/2;
  if ( v[j] < x ) {
    esq = j + 1;
  } else  if ( v[j] > x ) {
    drt = j - 1;
  } else {
    trobat = 1; break;
  }
}
if ( trobat == 1 ) {
  /* hi es  v[j] == x */
} else {
  /* no hi es */
}
```

Calen $O(\log_2(n))$ comparacions per a trobar l'element

Ordenació: selecció

Es busca el mínim i es posa al davant

```
for( i = 0; i< n-1; ++i) {
  sel = i;
  for( j = i+1; j<n; ++j) {
    if ( v[j] < v[sel] ) sel = j;
  }
  if ( i != sel ) {
    aux = v[i];
    v[i] = v[sel];
    v[sel] = aux;
  }
}
```

Ordenació: selecció

	0	1	2	3	4	5	6	7	8
	1	2	3	7	8	4	3	9	5
i=0	1	2	3	7	8	4	3	9	5
i=1	1	2	3	7	8	4	3	9	5
i=2	1	2	3	7	8	4	3	9	5
i=3	1	2	3	3	8	4	7	9	5
i=4	1	2	3	3	4	8	7	9	5
i=5	1	2	3	3	4	5	7	9	8
i=6	1	2	3	3	4	5	7	9	8
i=7	1	2	3	3	4	5	7	8	9

passada *i*: l'element que ha d'ocupar la posició *i* s'intercanvia amb el que l'ocupa

Ordenació: intercanvi (bombolla)

Si dos elements consecutius no estan en ordre, s'ordenen

```
for( i = 0; i< n-1; ++i) {
  for( j = n-1 ; j>i; --j) {
    if ( v[j] < v[j-1] )  {
      aux = v[j-1];
      v[j-1] = v[j];
      v[j] = aux;
    }
  }
}
```

Ordenació: bombolla

	0	1	2	3	4	5	6	7	8
	1	2	3	7	8	4	3	9	5
i=0	1	2	3	3	7	8	4	5	9
i=1	1	2	3	3	4	7	8	5	9
i=2	1	2	3	3	4	5	7	8	9
i=3	1	2	3	3	4	5	7	8	9
i=4	1	2	3	3	4	5	7	8	9
i=5	1	2	3	3	4	5	7	9	8
i=6	1	2	3	3	4	5	7	9	8
i=7	1	2	3	3	4	5	7	8	9

passada *i*: l'element que ha d'ocupar la posició *i* s'hi posa, descendint els "petits" per permuta d'elements consecutius desordenats. Fixeu-vos que continua fent passades malgrat estar ja ordenat

Ordenació: inserció lineal o directa

Un element es posa a lloc movent els més grans cap a darrera

```
for( i = 1; i< n; ++i) {
  aux = v[i];
  for( j = i-1 ; j>=0 && aux < v[j]; --j) {
    v[j+1] = v[j];
  }
  v[j+1] = aux;
}
```

Ordenació: inserció directa

	0	1	2	3	4	5	6	7	8
	1	2	3	7	8	4	3	9	5
i=1	1	2	3	7	8	4	3	9	5
i=2	1	2	3	7	8	4	3	9	5
i=3	1	2	3	7	8	4	3	9	5
i=4	1	2	3	7	8	4	3	9	5
i=5	1	2	3	4	7	8	3	9	5
i=6	1	2	3	3	4	7	8	9	5
i=7	1	2	3	3	4	5	7	8	9
i=8	1	2	3	3	4	5	7	8	9

passada i: $v[i]$ es posa a la seva posició en la part $[0,i]$

Ordenació: inserció binària

Un element es posa a lloc (cercant-lo de forma binària) i movent els més grans cap a darrera

```
for( i = 1; i< n; ++i) {
  aux = v[i];
  esq = 0; drt = i-1;
  while (esq <= drt) {
    j = (esq+drt)/2;
    if ( aux < v[j] ) { drt = j - 1; }
    else { esq = j + 1; }
  }
  for (j=i-1; j>=esq; --j) { v[j+1] = v[j]; }
  v[esq] = aux;
}
```

Ordenació: quicksort

El mètode **quicksort** també es basa en la idea "*divide and conquer*"

- Per a ordenar un vector:
- S'escull un element anomenat "pivot"
- Es reorganitza el vector al voltant del pivot: a la esquerra elements més petits i a la dreta més grans.
- S'aplica recursivament amb zona dels "petits" i amb la els 'grans", fins a arribar a zones de llargada 1, les quals ja estan ordenades

Ordenació: quicksort

La bondat de quicksort depèn de la tria de pivot: quan més igualtat en la mida de les 2 zones millor funciona.

En general és $O(n\log(n))$, i és el més ràpid conegut per a dades aleatòries. Si les les dades inicials són "dolentes" pot comportar-se com $O(n^2)$

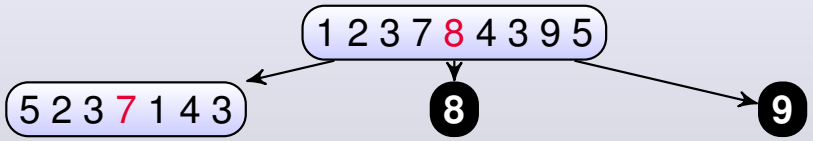
Ordenació: quicksort (funció recursiva)

```
void quicksort (double *v, int esq, int drt) {
  double aux;
  int i, k;
  if (esq >= drt ) return;
  /* pivot: element central; es mou a l'esquerra */
  i = (esq + drt) / 2;
  aux = v[esq]; v[esq] = v[i]; v[i] = aux;

  /* tots el petits cap a la part davantera */
  k = esq;
  for ( i = esq + 1; i<=drt; ++i) {
    if (v[i] < v[esq] ) {
      ++k;
      aux = v[k]; v[k]= v[i]; v[i] = aux;
    }
  }
```

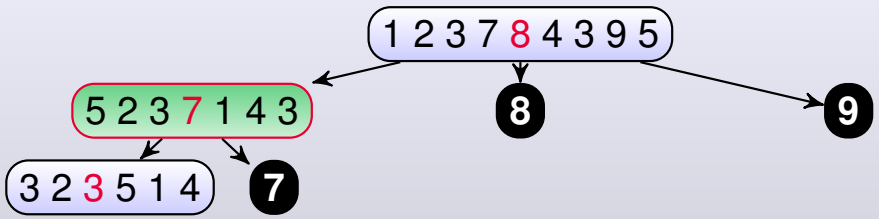
Ordenació: quicksort (cont. funció recursiva)

```
    /* el pivot al seu lloc */
    aux = v[esq]; v[esq] = v[k]; v[k] = aux;

    /* es fa primer la zona curta:
       estalvi recursivitat */
    if ( k - esq <= drt - k) {
      quicksort(v, esq, k - 1);
      quicksort(v, canvi + 1, drt);
    } else {
      quicksort(v, k + 1, drt);
      quicksort(v, esq, k - 1);
    }
    return;
}
```

Ordenació: quicksort (simulació)

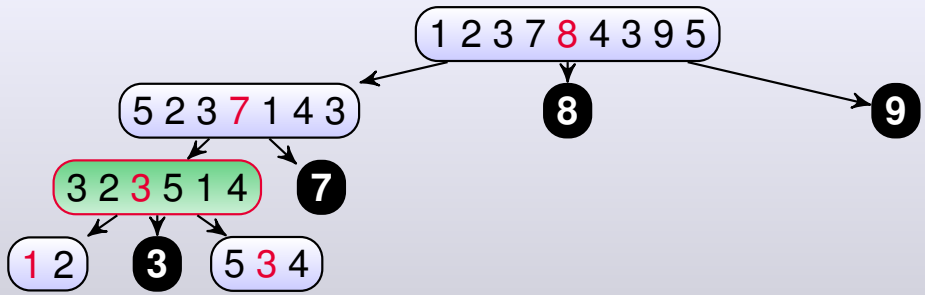


	0	1	2	3	4	5	6	7	8
inici	1	2	3	7	8	4	3	9	5
piv a 0	8	2	3	7	1	4	3	9	5
i=1 k=1	8	2	3	7	1	4	3	9	5
i=2 k=2	8	2	3	7	1	4	3	9	5
i=3 k=3	8	2	3	7	1	4	3	9	5
i=4 k=4	8	2	3	7	1	4	3	9	5
i=5 k=5	8	2	3	7	1	4	3	9	5
i=6 k=6	8	2	3	7	1	4	3	9	5
i=7 k=6	8	2	3	7	1	4	3	9	5
i=8 k=7	8	2	3	7	1	4	3	5	9
piv a lloc	5	2	3	7	1	4	3	8	9

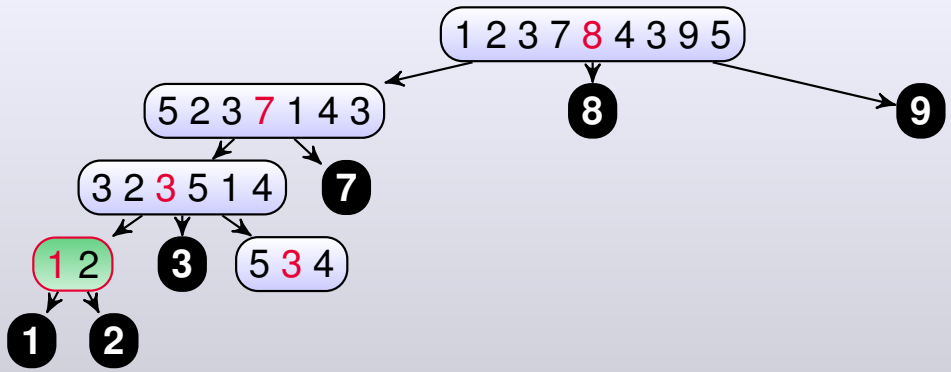


	0	1	2	3	4	5	6
inici	5	2	3	7	1	4	3
piv a 0	7	2	3	5	1	4	3
i=1 k=1	7	2	3	5	1	4	3
i=2 k=2	7	2	3	5	1	4	3
i=3 k=3	7	2	3	5	1	4	3
i=4 k=4	7	2	3	5	1	4	3
i=5 k=5	7	2	3	5	1	4	3
i=6 k=6	7	2	3	5	1	4	3
piv a lloc	3	2	3	7	1	4	7

Ordenació: quicksort (simulació)

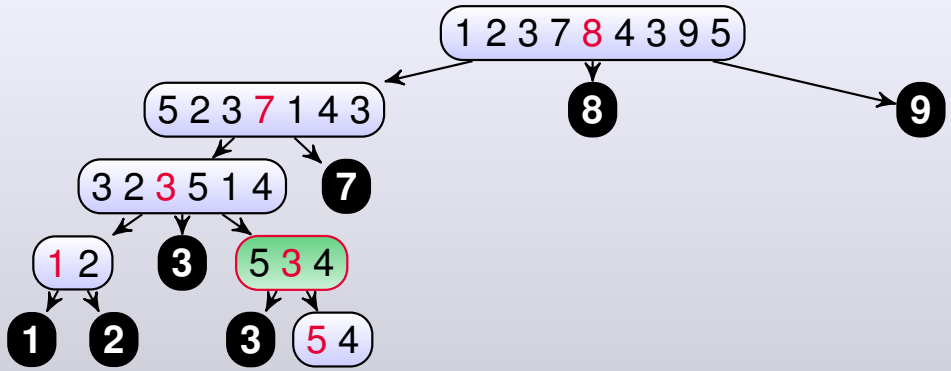


	0	1	2	3	4	5
inici	3	2	3	5	1	4
piv a 0	3	2	3	5	1	4
i=1 k=1	3	2	3	5	1	4
i=2 k=1	3	2	3	5	1	4
i=3 k=1	3	2	3	5	1	4
i=4 k=2	3	2	1	5	3	4
i=5 k=2	3	2	1	5	3	4
piv a lloc	1	2	3	5	3	4

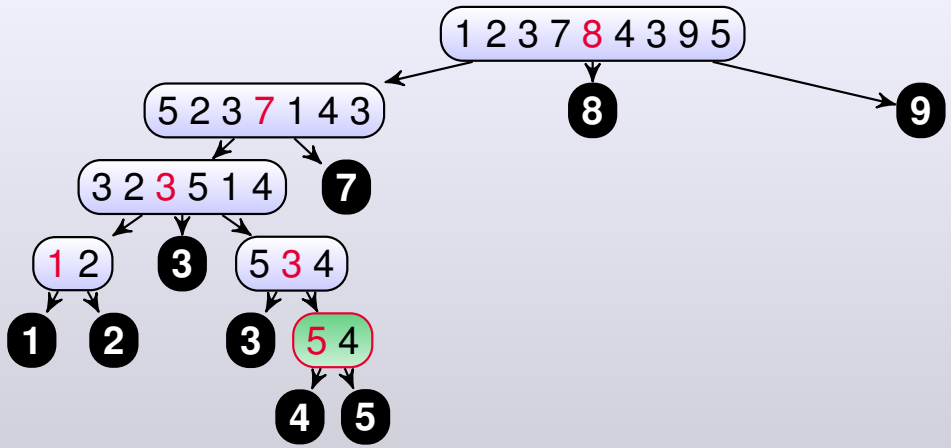


	0	1
inici	1	2
piv a 0	1	2
i=1 k=0	1	2
piv a lloc	1	2

Ordenació: quicksort (simulació)



	3	4	5
inici	5	3	4
piv a 0	3	5	4
i=4 k=3	3	5	4
i=5 k=3	3	5	4
piv a lloc	3	5	4

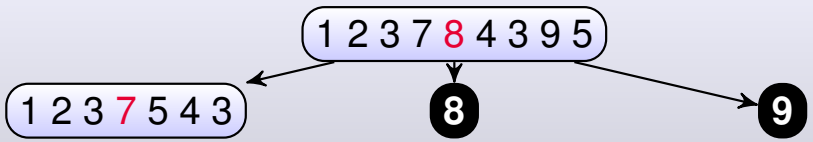


	4	5
inici	5	4
piv a 0	5	4
i=1 k=1	5	4
piv a lloc	4	5

Ordenació: quicksort (alternativa)

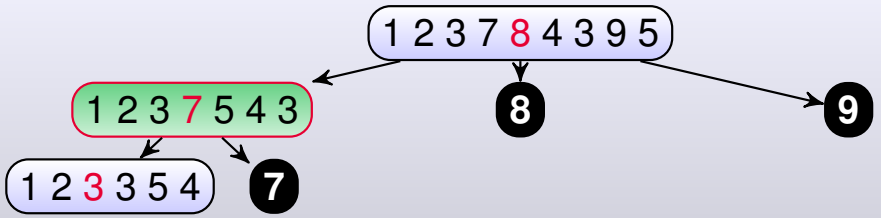
```
void quicksort (double *v, int esq, int drt) {
  double pivot;
  int i=esq, j=drt, aux;
  pivot = v[(esq + drt) / 2];
  while (i <= j ) {
    while (v[i] < pivot ) ++i;
    while (v[j] > pivot ) --j;
    if (i <= j ) {
      aux = v[i]; v[i] = v[j]; v[j] = aux;
      ++i; --j;
    }
  }
  if (esq < j ) { quicksort (v, esq, j); }
  if (i < drt ) { quicksort (v, i, drt); }
  return;
}
```

Ordenació: quicksort (alternativa, simulació)



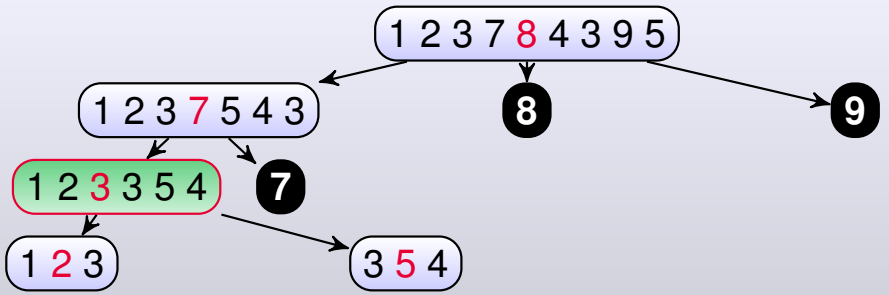
	0	1	2	3	4	5	6	7	8
inici	1	2	3	7	8	4	3	9	5
i=4 j=8	1	2	3	7	5	4	3	9	8
i=7 j=7	1	2	3	7	5	4	3	9	8
i=8 j=6	1	2	3	7	5	4	3	9	8

8



	0	1	2	3	4	5	6
inici	1	2	3	7	5	4	3
i=3 j=6	1	2	3	3	5	4	7
i=6 j=4	1	2	3	3	5	4	7

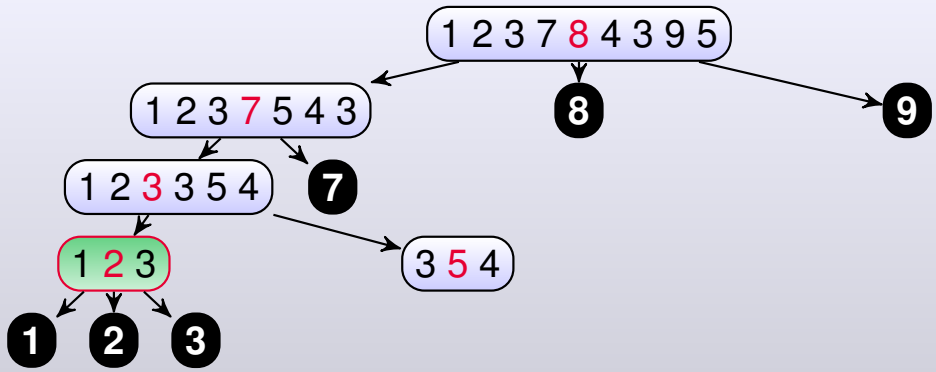
7

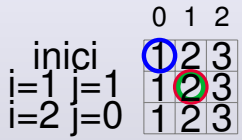


	0	1	2	3	4	5
inici	1	2	3	3	5	4
i=0 j=4	1	2	3	5	3	4
i=2 j=2	1	2	3	5	3	4
i=3 j=1	1	2	3	5	3	4

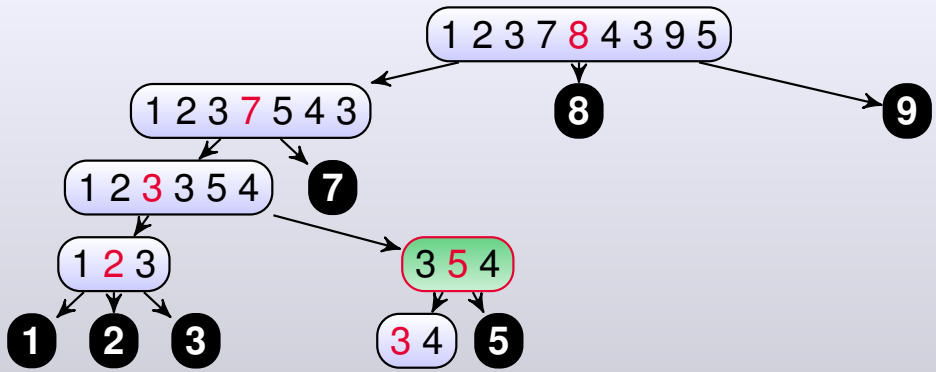
3

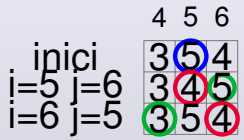
Ordenació: quicksort (alternativa, simulació)





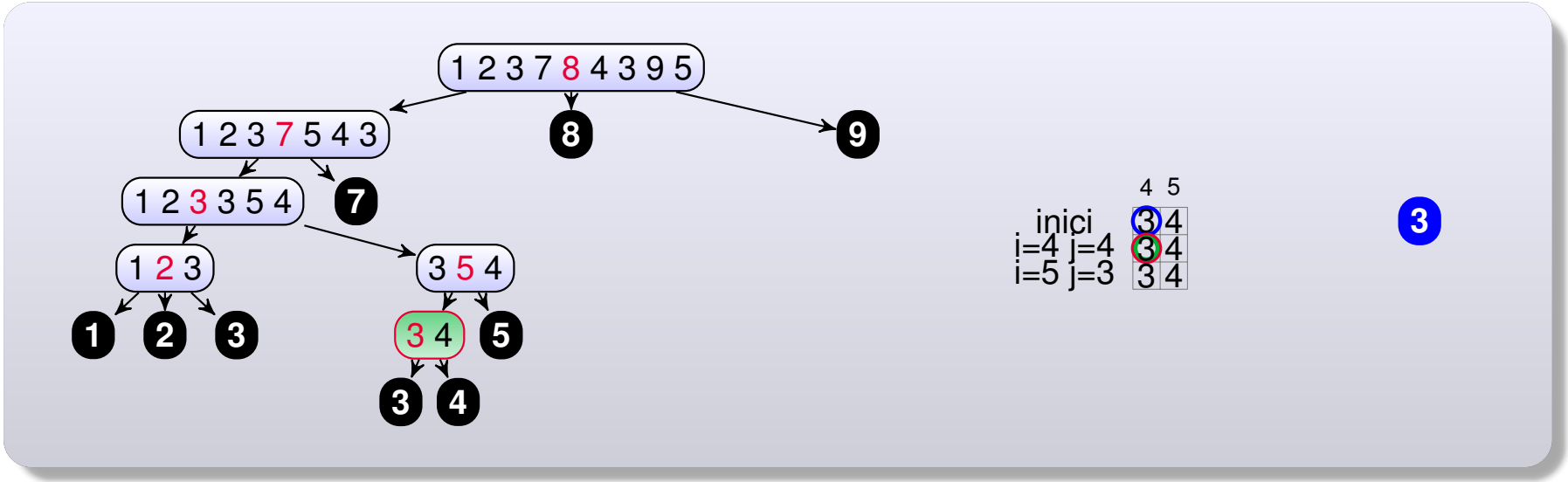
2





5

Ordenació: quicksort (alternativa, simulació)



Ordenació: quicksort (funció **qsort**)

Les funcions **quicksort** vistes estan escrites per a ordenar vectors de tipus simples, per als quals s'usa $<, >, \leq, \geq$.
La biblioteca **stdlib.h** ofereix una funció genèrica per a ordenar qualsevol mena d'entitats, només cal suministrar una funció per a poder-les comparar.

```
#include <stdlib.h>

void qsort(void *v, size_t n, size_t mida,
           int (*compar)(const void *, const void *));
```

Ordenació: quicksort (funció `qsort`)

```
void qsort(void *v, size_t n, size_t mida,
           int (*compar)(const void *, const void *));
```

`qsort` ordena de manera ascendent

- `v` adreça del vector a ordenar
- `n` dimensió del vector
- `mida` mida d'un element del vector
- `compar` adreça de la funció de comparar
  admet dos arguments, apuntadors a entitats a comparar
  si retorna un enter **negatiu**, el primer és **menor** que el segon
  si retorna un enter **positiu**, el primer és **major** que el segon
  si retorna **zero**, els dos elements són **iguals**

Els apuntadors són `void` ja que es pot usar per a qualsevol mena d'entitat definida en un programa.

Ordenació: Ús de qsort

```c
#include <stdio.h>
#include <stdlib.h>

#define SIGNE(x) ((x)>0?'+':'-')
typedef struct {
    int s;
    unsigned int num,den;
} frac;

int compFrac(const void *, const void *);
```

Ordenació: Ús de qsort

```
int main(void) {
  int n, j, k,m;
  frac *v;

  /* quantitat d'elements */
  scanf("%d", &n);
  if (n<1) return 1;

  v = (frac *) malloc(n*sizeof(frac));
  if (v == NULL) {
    printf("error de memoria!\n");
    return 1;
  }
```

Ordenació: Ús de qsort

```
/* lectura de dades  i normalitzacio signe */
for (j=0; j<n; j++) {
  v[j].s=1;
  scanf("%d/%d", &k, &m);
  if (k<0) { v[j].s=-v[j].s; v[j].num=-k;}
  else v[j].num=k;
  if (m<0) { v[j].s=-v[j].s; v[j].den=-m;}
  else v[j].den=m;
}
printf("llista␣inicial␣llegida:␣\n");
printf("␣␣␣i␣␣␣␣␣frac␣\n");
for (j=0; j<n; j++)
  printf("%5d␣%c%d/%d␣\n",
    j, SIGNE(v[j].s), v[j].num, v[j].den);
printf("\n");
```

Ordenació: Ús de qsort

```
  qsort(v, n, sizeof(frac), compFrac);

  printf("llista ordenada:\n");
  printf("   i      frac \n");
  for (j=0; j<n; j++)
    printf("%5d %c%d/%d \n",
      j, SIGNE(v[j].s), v[j].num, v[j].den);
  printf("\n");
  free(v);
  return 0;
}
```

Ordenació: Ús de qsort

```
/* funcio de comparacio segons l'ordre natural

a/b < c/d  -> ad < bc -> ad-bc < 0

*/
int compFrac(const void *p, const void *q) {
  frac *r =(frac *)p, *t = (frac*)q;
  if (r->s > t->s) return 1;          /* p > q */
  else if (r->s < t->s) return -1; /* q > p */
  return (r->s)*(r->num*t->den - t->num*r->den);
}
```

Ordenació: mergesort

El mètode **mergesort** es basa en el principi "*divide and conquer*" ("*divide et impera*" atribuïda a Cèsar)

- Per a ordenar un vector:
- S'ordenen les dues meitats
- Es fusionen les dues meitats (ordenadament)
- S'aplica recursivament, fins a arribar a elements de llargada 1, els quals ja estan ordenats

Aquesta seria una implementació "top-down", però es podria fer "bottom-up": anant doblant les dimensions
Atenció: usa vectors auxiliars per a fer les fusions

Ordenació: mergesort (funció recursiva)

```
void mergeSort(double *arr, int esq, int drt) {
  int m = esq+(drt-esq)/2;
  if (esq < drt) {
    mergeSort(arr, esq, m);
    mergeSort(arr, m+1, drt);
    fusio(arr, esq, m, drt);
  }
  return;
}
```

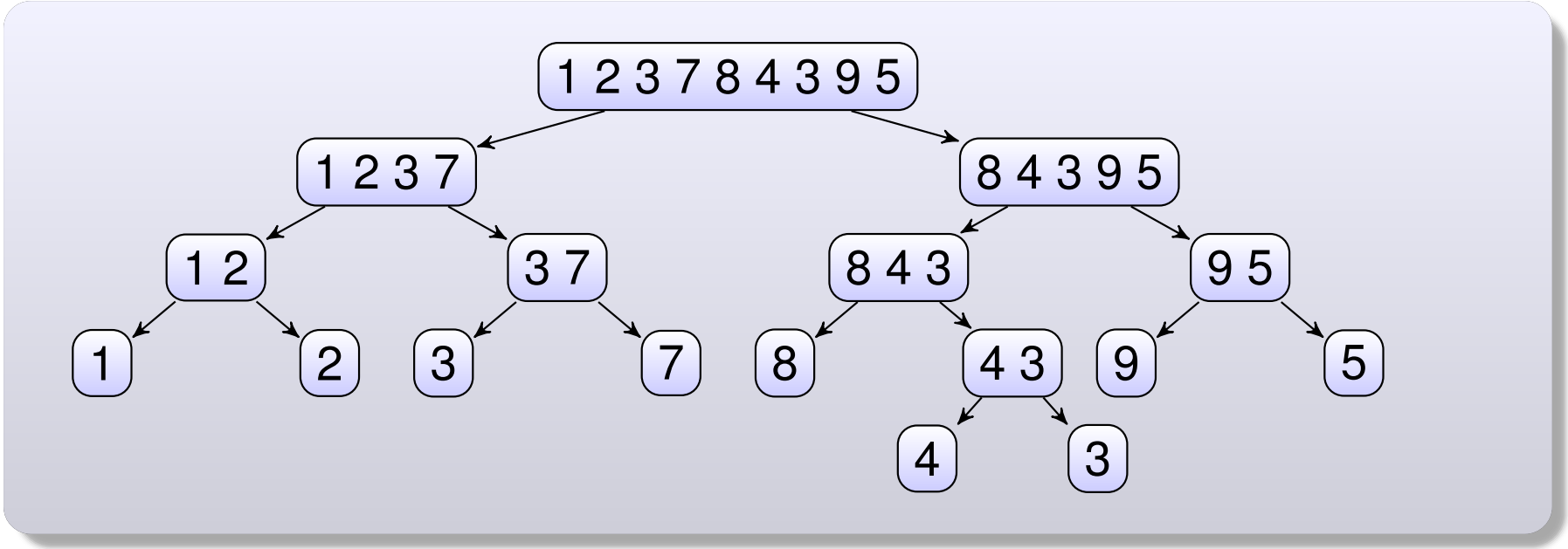
Ordenació: mergesort (funció de fusió)

```
void fusio(double *arr, int esq, int m, int drt) {
  int i, j, k, n1 = m - esq + 1, n2 = drt - m;
  double *L , *R;
  L= (double *) malloc (n1*sizeof(double));
  R= (double *) malloc (n2*sizeof(double));
  if (L == NULL || R == NULL ) {
    printf("error mem L=%d,m=%d,R=%d\n",esq,m,drt);
    exit (1);
  }
  /* Copia de cada part en auxiliar */
  for (i = 0; i < n1; ++i) L[i] = arr[esq + i];
  for (i = 0; i < n2; ++i) R[i] = arr[m + 1+ i];
```

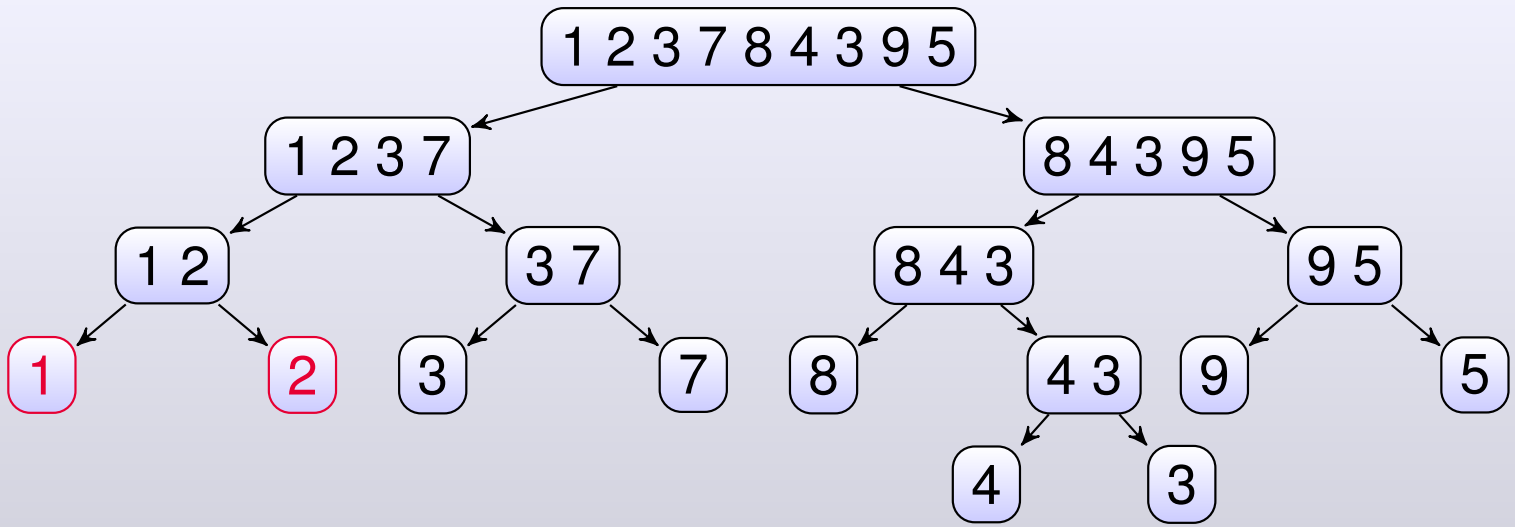
Ordenació: mergesort (funció de fusió)

```
  k = esq;
  /* fins que s'acabi la part mes curta */
  for (i=j=0; i < n1 && j < n2; ++k) {
    /* posa el mes petit al vector */
    if (L[i] <= R[j]) { arr[k] = L[i++]; }
    else { arr[k] = R[j++]; }
  }
  /* els sobrants de la llarga es copien */
  while (i < n1) arr[k++] = L[i++];
  while (j < n2) arr[k++] = R[j++];
  free (L); free (R);
  return;
}
```

Ordenació: mergesort (simulació)

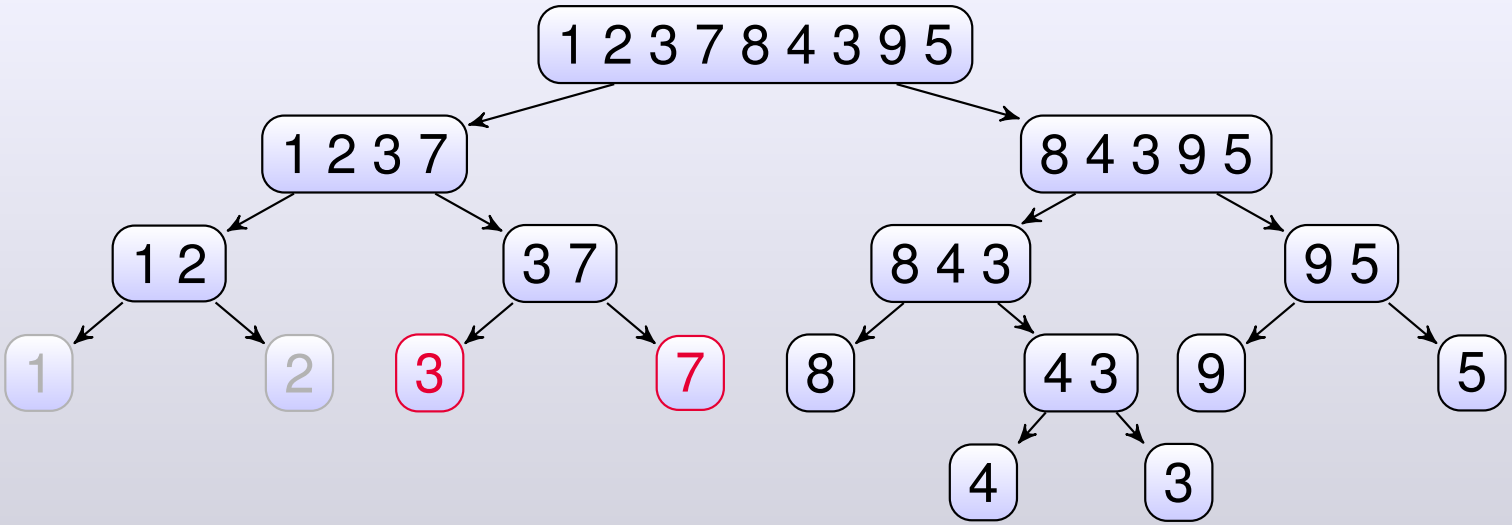


Ordenació: mergesort (simulació)



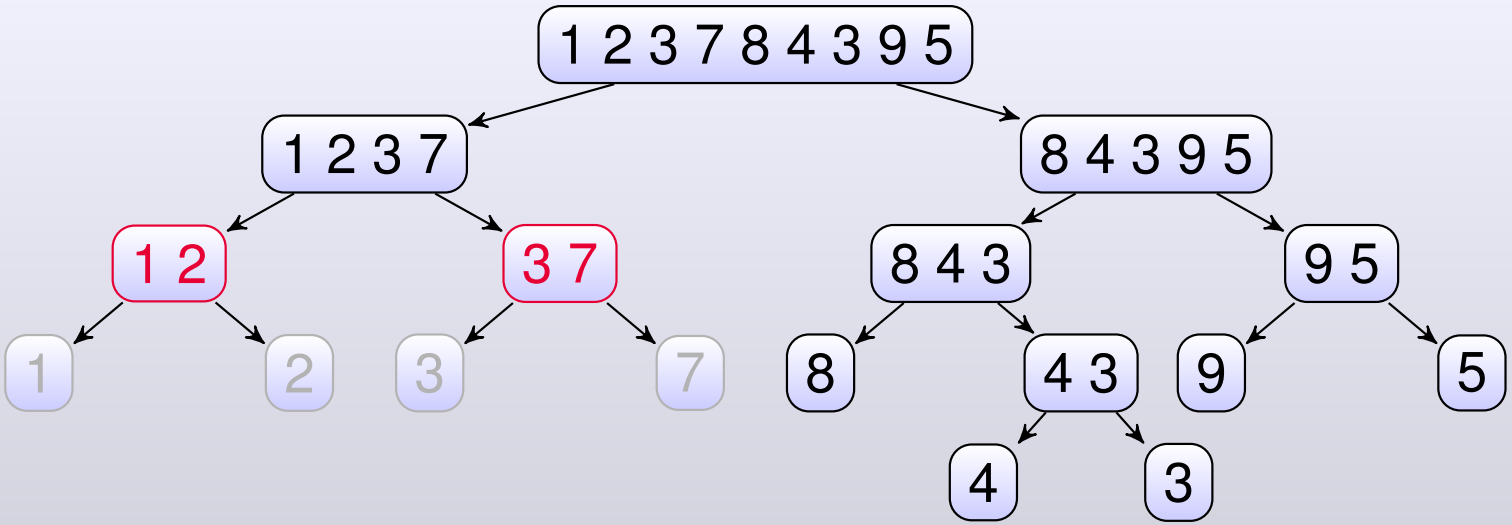
1 i 2 ja estan; es fusionen

Ordenació: mergesort (simulació)



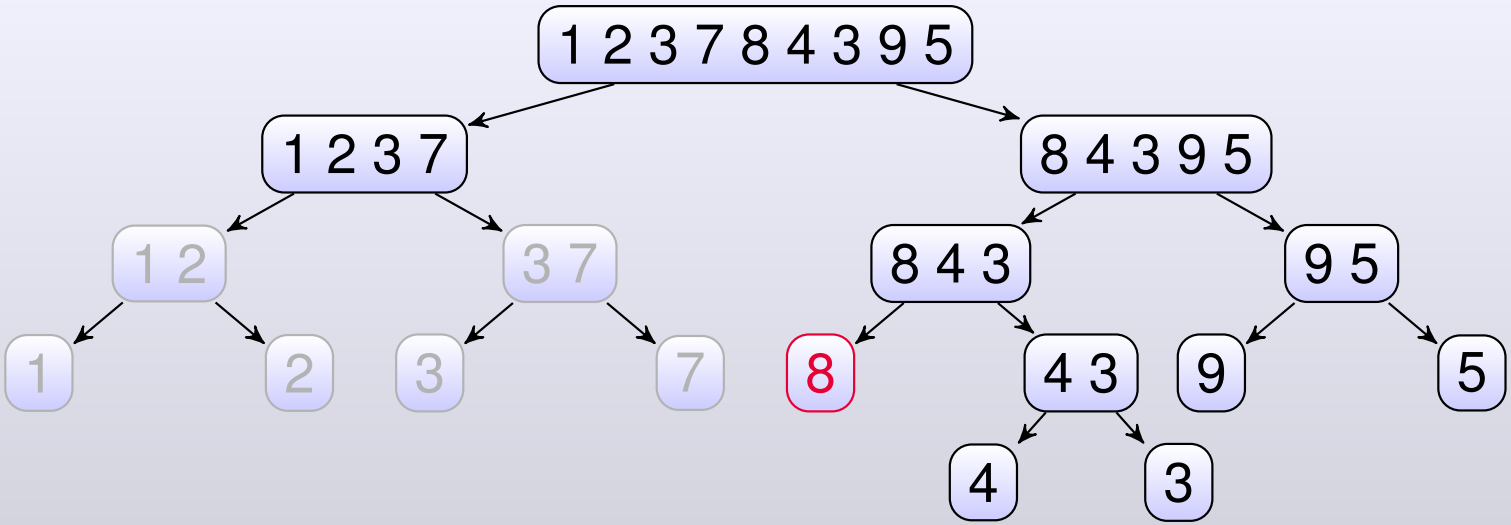
3 i 7 ja estan; es fusionen

Ordenació: mergesort (simulació)



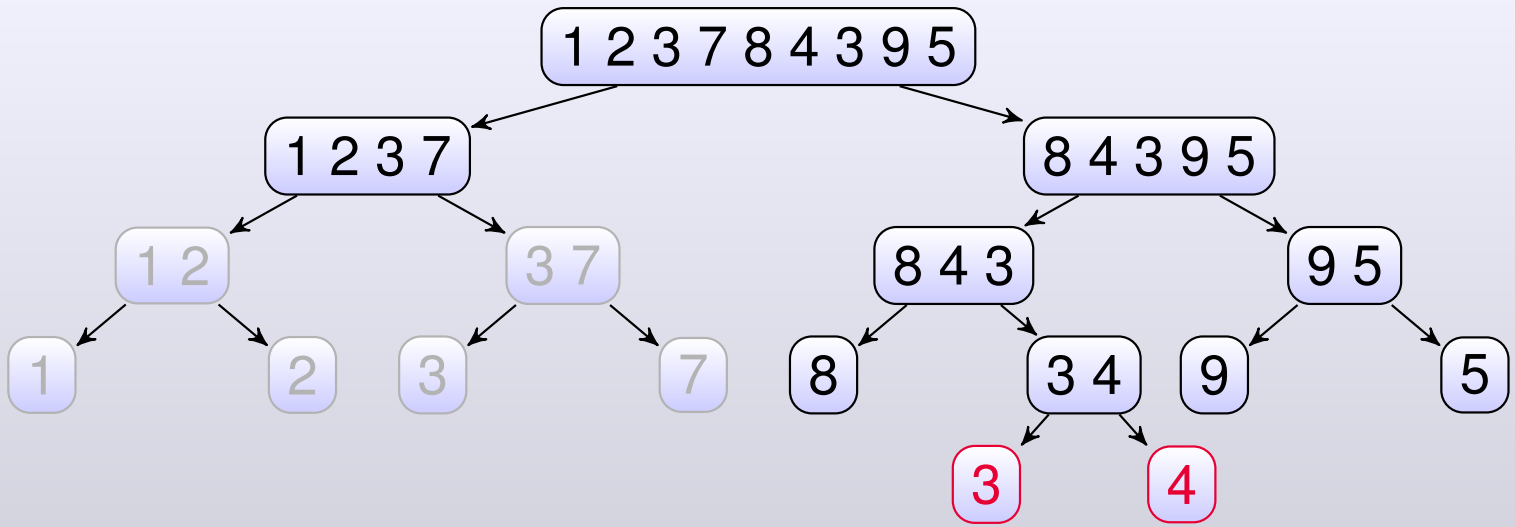
1,2 i 3,7 ja estan; es fusionen

Ordenació: mergesort (simulació)



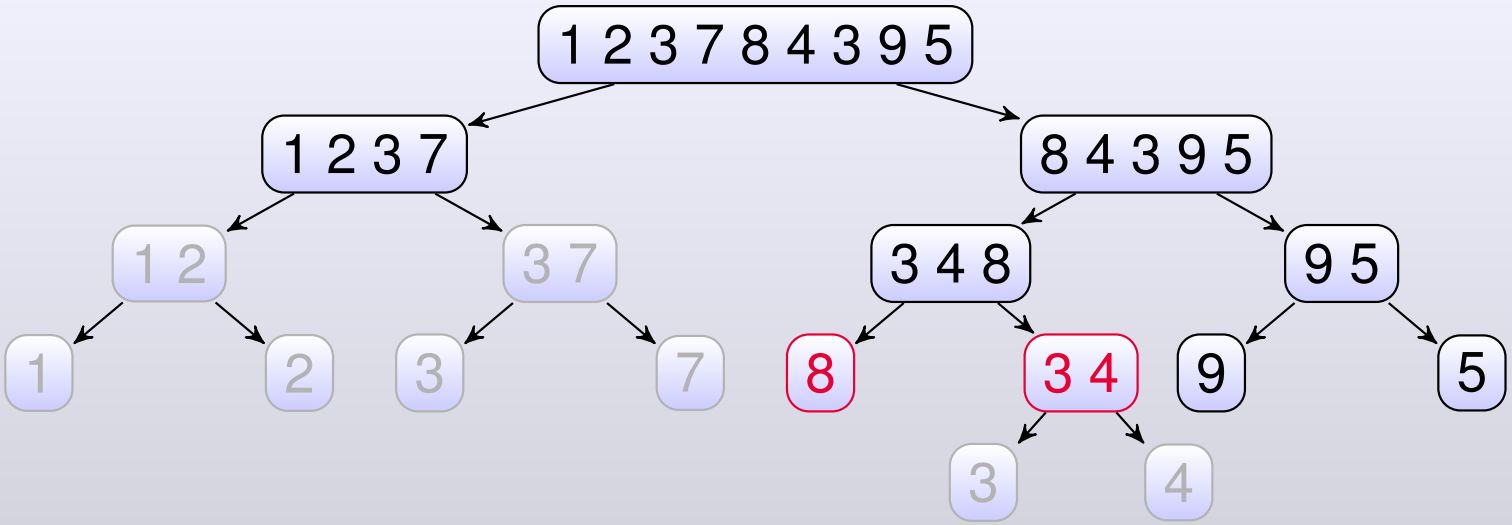
8 ja està;

Ordenació: mergesort (simulació)



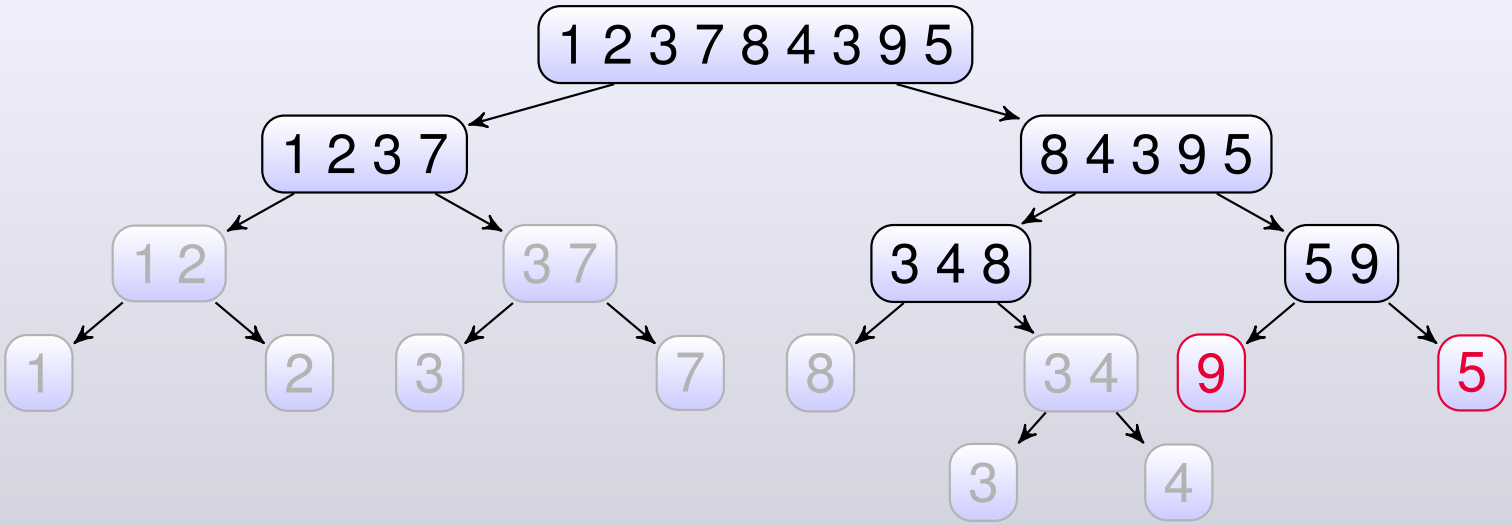
4 i 3 estan; es fusionen (i queden canviats)

Ordenació: mergesort (simulació)



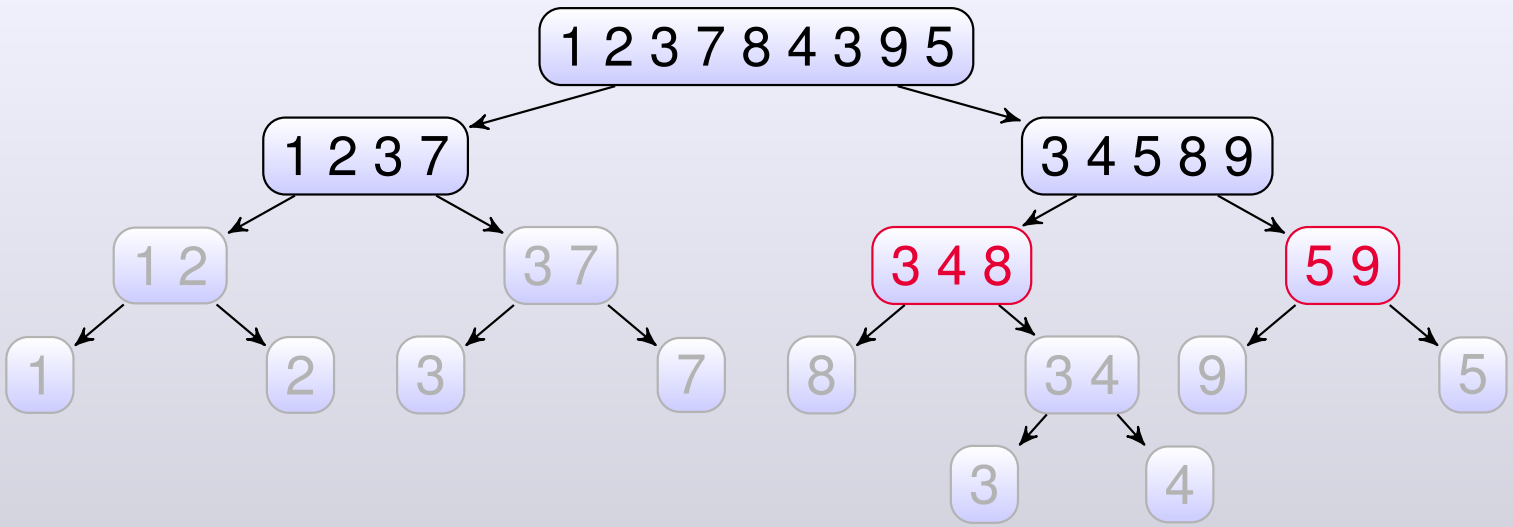
8 i 3,4 ja estàn; es fusionen (ordenant-se)

Ordenació: mergesort (simulació)



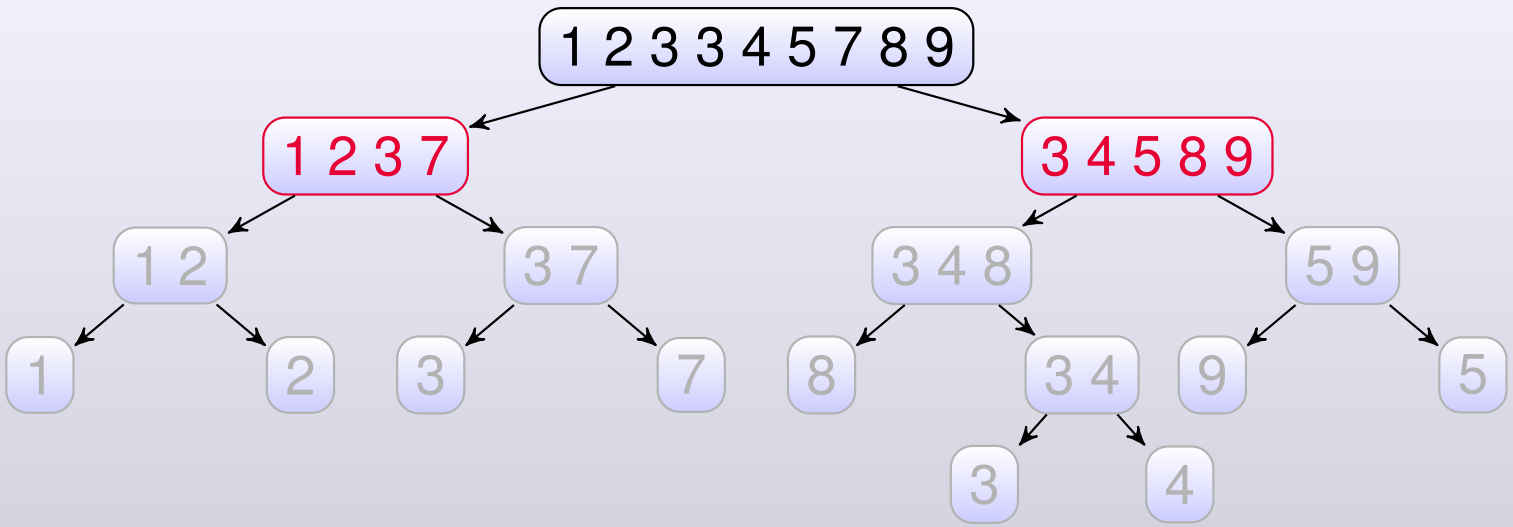
9 i 5 ja estàn; es fusionen (ordenant-se)

Ordenació: mergesort (simulació)



3,4,8 i 5,9 ja estàn; es fusionen (ordenant-se)

Ordenació: mergesort (simulació)



1,2,3,7 i 3,4,5,8,9 ja estàn; es fusionen (ordenant-se)

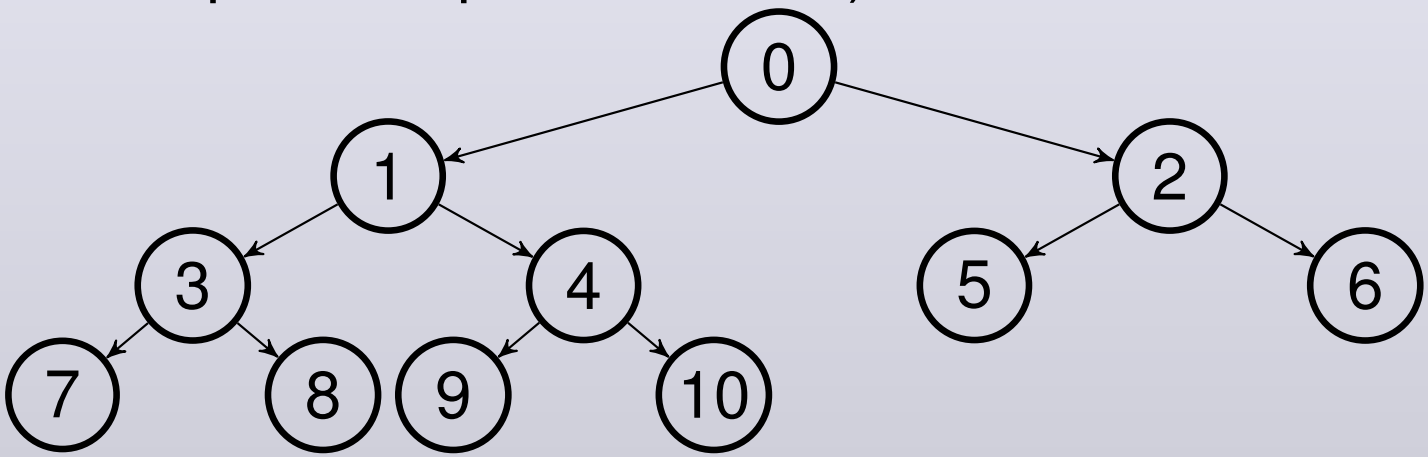
Ordenació: heapsort

El mètode **heapsort** és una millora de selecció directa (aprofita comparacions fetes quan es busca el mínim)
Per a introduir el concepte: *Arbre binari gairebé ple ordenat* s'usarà nomenclatura de "grafs"

- Graf orientat : Sistema finit de nodes i d'arestes (connexions) orientades (té importància el sentit de la connexió). S'introdueix la relació d'origen a final de la connexió com *pare* a *fill*
- Arbre: Graf orientat amb un únic node (**arrel**) sense pare, on cada node té un pare (excepte l'arrel) i una quantitat finita de fills (si no té fills se'n diu **fulla**)

Ordenació: heapsort

- Arbre binari: cada node té com a molt dos fills
- Arbre ordenat: els fills d'un mateix node (*germans*) estàn ordenats (en el sentit que es poden distingir) d'esquerra a dreta.
- Arbre binari ordenat gairebé complet: el que té ocupats tots els nodes (exceptuant l'últim nivell que estarà ple d'esquerra a dreta).



Ordenació: heapsort

- Un arbre binari *complet* de N nivells té $2^N - 1$ nodes
- és pot identificar amb un vector

$$v_0, v_1, \ldots, v_{2^N-2}$$

i si és gairebé complet falten els darrers elements de l'últim nivell.

Ordenació: heapsort

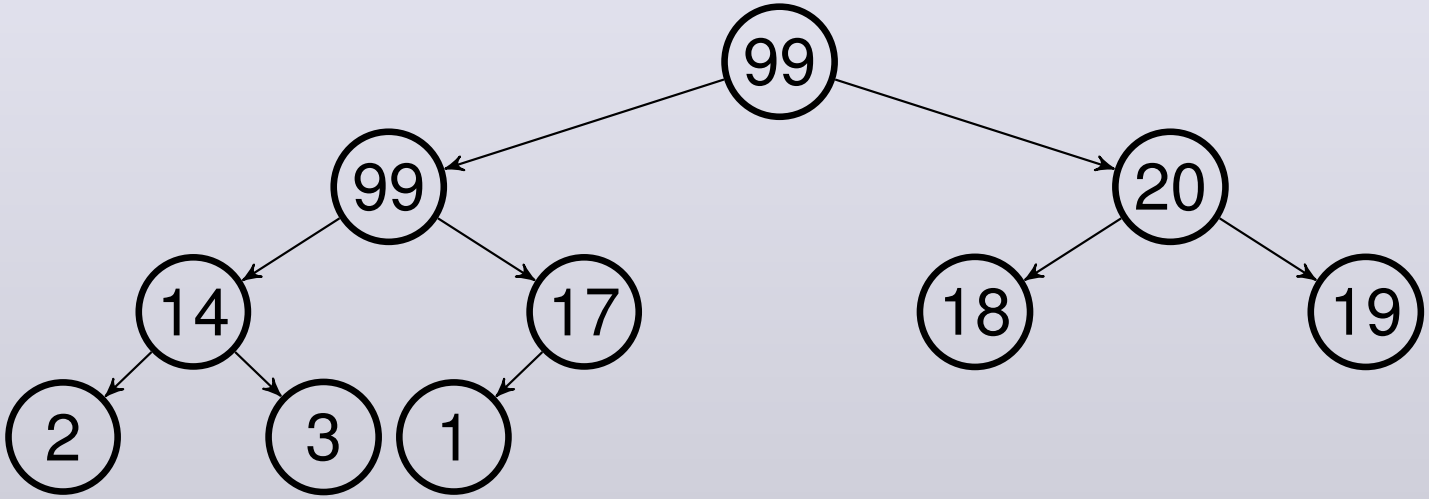
Propietats d'un vector identificat amb "ABOGC"

- fills del node j:
  $2j+2 < n$ són els que tenen índexs $2j+1, 2j+2$
  $2j+1 = n-1$ és el que té índex $2j+1$
  $2j+1 > n-1$ no té fills
- índex del pare del node $j \neq 0$: $\lfloor j/2 \rfloor$ (div. entera)
  el pare del node 0 (arrel) és inexistent
- node j té fills si $j \leq \lfloor (n-1)/2 \rfloor$
- el darrer node amb fills té índex $j = \lfloor (n-1)/2 \rfloor$
- són fulles els nodes j tals que
  $\lfloor (n-1)/2 \rfloor + 1 \leq j < n$
- indexs de nivell k: $\left\{ j | 2^k - 1 \leq j \leq \min(2^{k+1}-2, n-1) \right\}$

Ordenació: heapsort

Jerarquia guanyadora (perdedora): és un "ABOGC" tal que cada node $j = 1, \ldots, n$ compleix **alguna** d'aquestes condicions:

- $j > \lfloor (n-1)/2 \rfloor$ (i no té fills)
- $2j + 1 = n$ i $v_j \geq v_{2j+1}$ (pare $\geq$ ($\leq$) únic fill)
- $2j + 2 \leq n$ i $v_j \geq v_{2j+1}$ i $v_j \geq v_{2j+2}$ (pare $\geq$ ($\leq$) 2 fills)



Ordenació: heapsort

El mètode **heapsort** agafa un vector desordenat:

- reordena per a estar en format jerarquia guanyadora: a l'arrel hi haurà l'element més gran
- permuta l'arrel amb l'últim element ($v_0 \leftrightarrow v_{n-1}$) i s'oblida de v_{n-1}
- ara l'arrel "*espatlla*" la jerarquia guanyadora, cal reorganitzar el vector (de dimensió $n-1$)
- s'itera el procediment fins a tenir dimensió 1

Ordenació: heapsort

Algorisme **heapsort**: (vector $(v_0, v_1, \ldots, v_{n-1})$)

1. "*contrucció*" de la jerarquia guanyadora:

   $$\forall i = \lfloor n/2 \rfloor - 1, \lfloor n/2 \rfloor - 2, \ldots, 0 \quad \text{ajustar}(v, i, n-1)$$

   Es comença a fer al primer node amb fills i es va pujant cap a l'arrel

2. intercanvis i reorganització:

   $$\forall i = n-1, n-2, \ldots, 1 \quad \{v_0 \leftrightarrow v_i; \quad \text{ajustar}(v, 0, i-1)\}$$

Ordenació: heapsort

```
for( i = n/2-1; i>=0; --i) {
  ajustar(v,i,n-1);
}
for( i = n-1; i>0; --i) {
  aux = v[0];
  v[0] = v[i];
  v[i] = aux;
  ajustar(v,0,i-1);
}
```

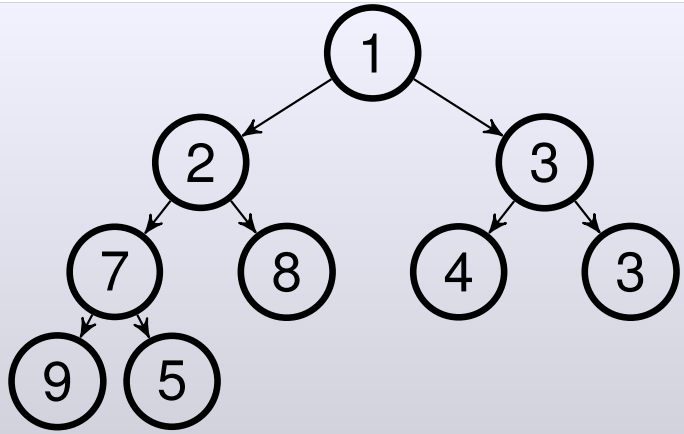
Ordenació: heapsort

```
void ajustar ( double *v, int esq, int drt) {
  double aux;
  int m = 2*esq+1; /* 1r fill */
  while ( m <= drt ) {/* mentre estem a l'arbre */
    /* si hi ha 2n fill i es major, l'agafem */
    if ( m+1 <= drt && v[m] < v[m+1])  m++;
    if ( v[m] > v[esq]) {
    /* si el pare es major que el fill, permutem */
      aux = v[esq]; v[esq] = v[m]; v[m] = aux;
      /* nou origen del subarbre */
      esq = m; m = 2 * esq+1;
    } else  /* te 2 fills mes petits, parem */
        break;
  }
  return;
}
```

Ordenació: heapsort (simulació)

Creació de la jerarquia guanyadora:

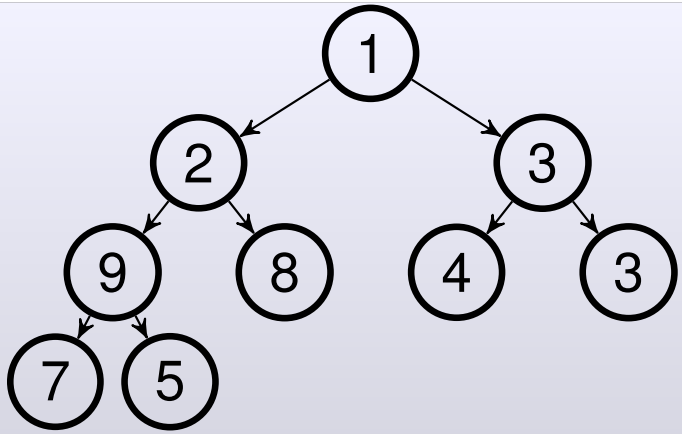
0	1	2	3	4	5	6	7	8
1	2	3	7	8	4	3	9	5



Ordenació: heapsort (simulació)

Creació de la jerarquia guanyadora:

	0	1	2	3	4	5	6	7	8
	1	2	3	7	8	4	3	9	5
i=3	1	2	3	9	8	4	3	7	5

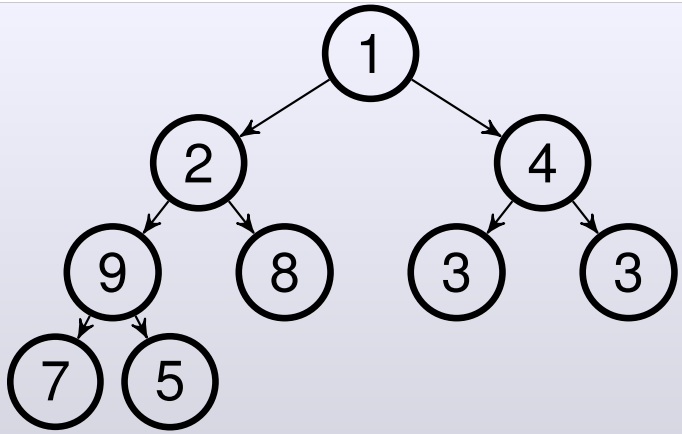


El subarbre del "9" és jerarquia guanyadora

Ordenació: heapsort (simulació)

Creació de la jerarquia guanyadora:

	0	1	2	3	4	5	6	7	8
	1	2	3	7	8	4	3	9	5
	1	2	3	9	8	4	3	7	5
i=2	1	2	4	9	8	3	3	7	5

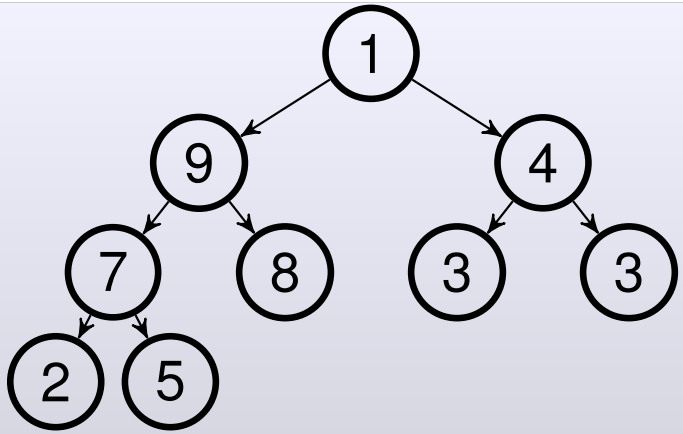


Els subarbres del "9" i del "4" són jerarquies guanyadores

Ordenació: heapsort (simulació)

Creació de la jerarquia guanyadora:

	0	1	2	3	4	5	6	7	8
	1	2	3	7	8	4	3	9	5
	1	2	3	9	8	4	3	7	5
	1	2	4	9	8	3	3	7	5
i=1	1	9	4	7	8	3	3	2	5

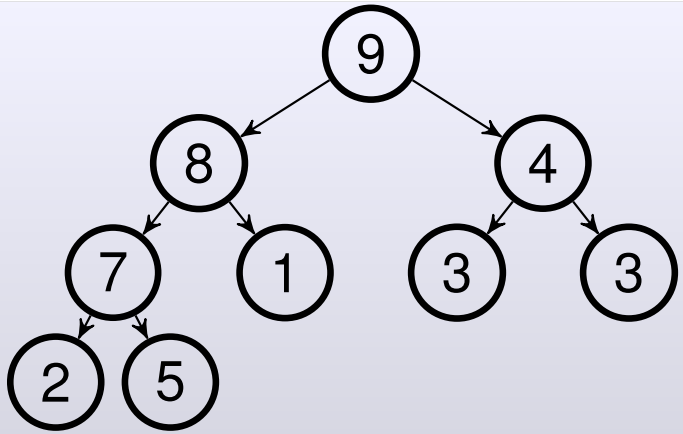


Els subarbres del "9" i del "4" són jerarquies guanyadores

Ordenació: heapsort (simulació)

Creació de la jerarquia guanyadora:

	0	1	2	3	4	5	6	7	8
	1	2	3	7	8	4	3	9	5
	1	2	3	9	8	4	3	7	5
	1	2	4	9	8	3	3	7	5
	1	9	4	2	8	3	3	7	5
i=0	9	8	4	7	1	3	3	2	5

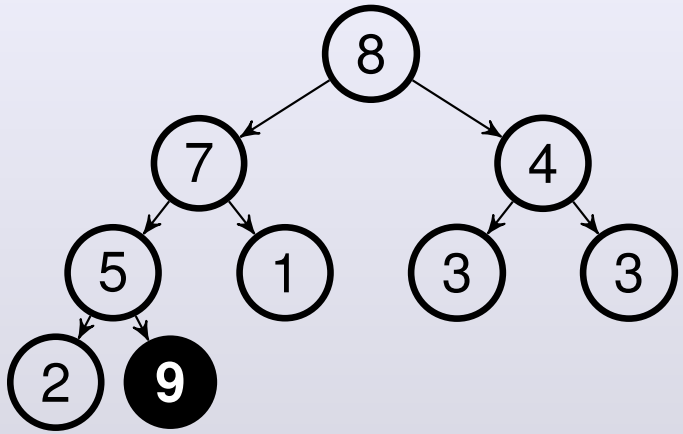


L'arbre és jerarquia guanyadora

Ordenació: heapsort (simulació)

Intercanvi amb arrel i reorganització:

	0	1	2	3	4	5	6	7	8
	9	8	4	7	1	3	3	2	5
i=8	8	7	4	5	1	3	3	2	9

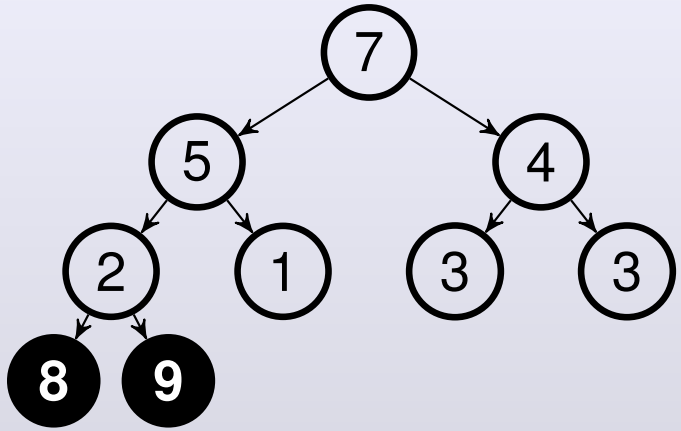


El "9" al final i el "5" s'enfonsa fins a ser jerarquia guanyadora

Ordenació: heapsort (simulació)

Intercanvi amb arrel i reorganització:

	0	1	2	3	4	5	6	7	8
	9	8	4	7	1	3	3	2	5
	8	7	4	5	1	3	3	2	9
i=7	7	5	4	2	1	3	3	8	9

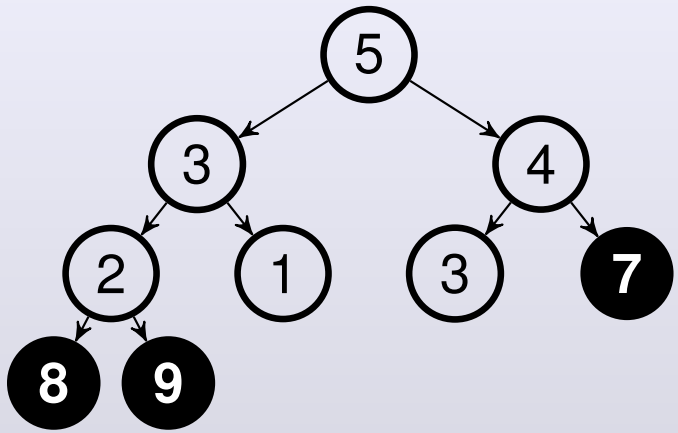


El "8" al final i el "2" s'enfonsa fins a ser jerarquia guanyadora

Ordenació: heapsort (simulació)

Intercanvi amb arrel i reorganització:

	0	1	2	3	4	5	6	7	8
	9	8	4	7	1	3	3	2	5
	8	7	4	5	1	3	3	2	9
	7	5	4	2	1	3	3	8	9
i=6	5	3	4	2	1	3	7	8	9

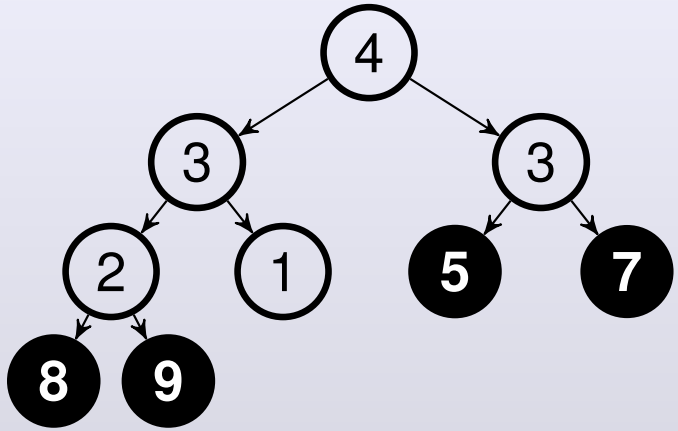


El "7" al final i el "3" s'enfonsa fins a ser jerarquia guanyadora

Ordenació: heapsort (simulació)

Intercanvi amb arrel i reorganització:

	0	1	2	3	4	5	6	7	8
	9	8	4	7	1	3	3	2	5
	8	7	4	5	1	3	3	2	9
	7	5	4	2	1	3	3	8	9
	5	3	4	2	1	3	7	8	9
i=5	4	3	3	2	1	5	7	8	9

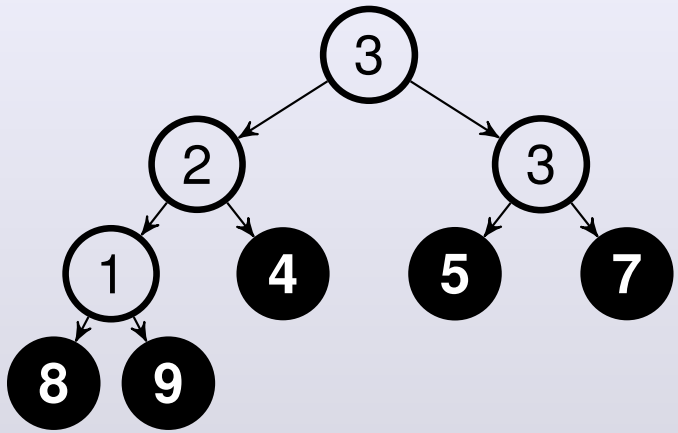


El "5" al final i el "3" s'enfonsa fins a ser jerarquia guanyadora

Ordenació: heapsort (simulació)

Intercanvi amb arrel i reorganització:

	0	1	2	3	4	5	6	7	8
	9	8	4	7	1	3	3	2	5
	8	7	4	5	1	3	3	2	9
	7	5	4	2	1	3	3	8	9
	5	3	4	2	1	3	7	8	9
	4	3	3	2	1	5	7	8	9
i=4	3	2	3	1	4	5	7	8	9

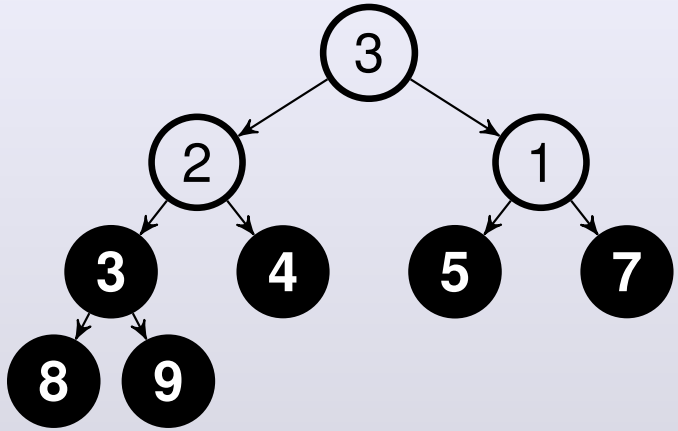


El "4" al final i l'"1" s'enfonsa fins a ser jerarquia guanyadora

Ordenació: heapsort (simulació)

Intercanvi amb arrel i reorganització:

	0	1	2	3	4	5	6	7	8
	9	8	4	7	1	3	3	2	5
	8	7	4	5	1	3	3	2	9
	7	5	4	2	1	3	3	8	9
	5	3	4	2	1	3	7	8	9
	4	3	3	2	1	5	7	8	9
	3	2	3	1	4	5	7	8	9
i=3	3	2	1	3	4	5	7	8	9

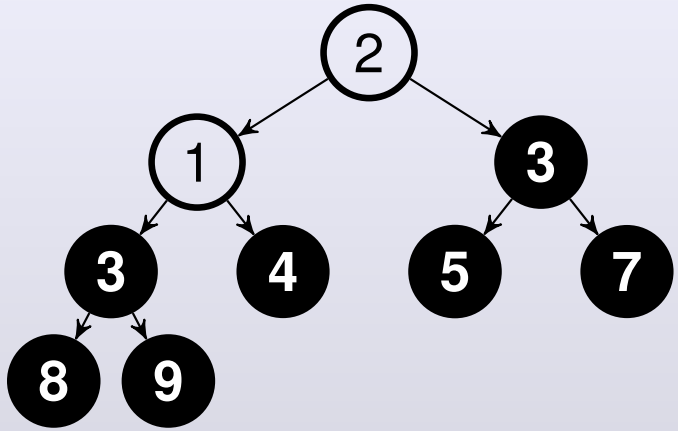


El "3" al final i l'"1" s'enfonsa fins a ser jerarquia guanyadora

Ordenació: heapsort (simulació)

Intercanvi amb arrel i reorganització:

	0	1	2	3	4	5	6	7	8
	9	8	4	7	1	3	3	2	5
	8	7	4	5	1	3	3	2	9
	7	5	4	2	1	3	3	8	9
	5	3	4	2	1	3	7	8	9
	4	3	3	2	1	5	7	8	9
	3	2	3	1	4	5	7	8	9
	3	2	1	3	4	5	7	8	9
i=2	2	1	3	3	4	5	7	8	9

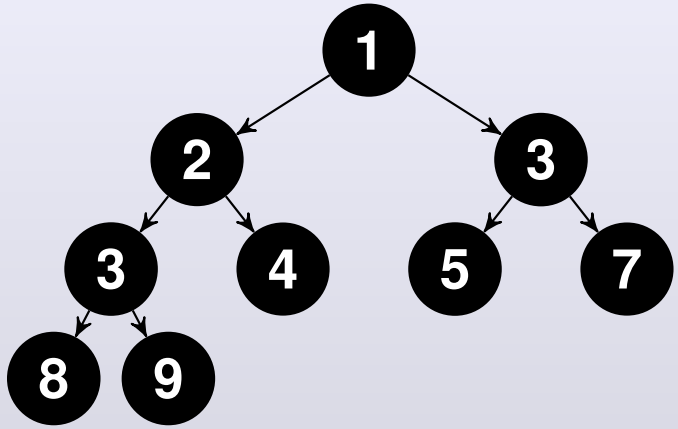


El "3" al final i l'"1" s'enfonsa fins a ser jerarquia guanyadora

Ordenació: heapsort (simulació)

Intercanvi amb arrel i reorganització:

	0	1	2	3	4	5	6	7	8
	9	8	4	7	1	3	3	2	5
	8	7	4	5	1	3	3	2	9
	7	5	4	2	1	3	3	8	9
	5	3	4	2	1	3	7	8	9
	4	3	3	2	1	5	7	8	9
	3	2	3	1	4	5	7	8	9
	3	2	1	3	4	5	7	8	9
	2	1	3	3	4	5	7	8	9
i=1	1	2	3	3	4	5	7	8	9



Es canvien el "2" i l'"1" . S'ha acabat