Arbres
Programació 2
Facultat d'Informàtica d'Informàtica, UPC

Professorat de PRO2

Tardor 2022

- Col·laboracions (en ordre alfabètic): Juan Luis Esteban, Ricard Gavaldà, Conrado Martínez, Fernando Orejas
- Aquestes transparències **no** substitueixen els apunts de l'assignatura, els complementen

Part I

[redacted]

Principis de disseny recursiu

Volem implementar recursivament una funció

```
// Pre: propietat satisfeta per x
// Post: la funció retorna un valor F(x)
tipus_sortida F(tipus_entrada x);
```

o un procediment

```
// Pre: propietat satisfeta per x = X
// Post: res compleix una certa propietat en termes d'X
void F(T1 x, T2& res);
```

Principis de disseny recursiu

N.B. `x` i `res` poden ser més d'un paràmetre; en el cas dels procediments podem tenir paràmetres d'entrada/sortida:

```
// Pre: propietat satisfeta per x = X
// Post: x = X' compleix una certa propietat en termes
//       del seu valor original X
void F(T1& x)
```

Principis de disseny recursiu

Cal identificar:

- Un o més casos base: Valors de paràmetres en què podem satisfer la Post amb càlculs directes
- Un o més casos recursius: Valors de paràmetres en què podem satisfer la Post si tinguessim el resultat per a alguns paràmetres x' "més petits" que x

Estratègia

1. Triar una funció de "mida" $|x|$ dels paràmetres x tal que
   - $|x| \leq 0 \implies$ som en un cas base
   - les crides recursives es fan amb paràmetres x' amb $|x'| < |x|$
   - ha de ser sempre un enter: per tot x, $|x| \in \mathbb{Z}$

   Fonament: tota seqüència decreixent d'enters no negatius és finita
2. Transformar la definició rebuda del que volem calcular en una definició recursiva (si no ho és d'entrada)

Correctesa d’un algorisme recursiu

A demostrar: Amb tot valor x dels paràmetres que satisfaci Pre,

- l’algorisme acaba - nombre finit de crides recursives
- i acaba satisfent $Post(x)$

Acabament: nombre finit de crides recursives

Fonament:

Tota seqüència decreixent de nombres enters no negatius és finita

Acabament: nombre finit de crides recursives

Fonament:

Tota seqüència decreixent de nombres enters no negatius és finita

Formalització:

- Triem una funció de mida $|\cdot|$ dels paràmetres que sempre té valor enter
- Demostrem: Si $|x| \leq 0$ l'algorisme tracta x amb un cas base $\Longrightarrow$ cap crida recursiva
- Demostrem: Cada crida recursiva fa decrèixer la mida dels paràmetres, i.e., si la funció `F` amb paràmetre x fa la crida recursiva $\texttt{F}(x')$ llavors $|x'| < |x|$

Correctesa d'un algorisme recursiu

- A demostrar: Si el paràmetre x satisfà la precondició llavors el resultat satisfà la postcondició (una funció d'x)
- Quan x és un cas base ($|x| \leq 0$, no hi ha recursió): és un cas senzill i directe.

Correctesa d'un algorisme recursiu

- Si x no és un cas base ($|x| > 0$), apliquem l'hipòtesi d'inducció:

  H.I. = "Si x' compleix la precondició ($\mathtt{Pre}(x')$ és cert) i $|x'| < |x|$ llavors l'algorisme acaba en temps finit i es compleix la postcondició ($\mathtt{Post}(x')$)"

Correctesa d'un algorisme recursiu

- Si x no és un cas base ($|x| > 0$), apliquem l'hipòtesi d'inducció:

  H.I. = "Si x' compleix la precondició ($\mathtt{Pre}(x')$ és cert) i $|x'| < |x|$ llavors l'algorisme acaba en temps finit i es compleix la postcondició ($\mathtt{Post}(x')$)"

- Hem de demostrar que qualsevol crida recursiva $\mathtt{F}(x')$ quan x no és un cas base ($|x| > 0$) compleix: 1) $\mathtt{Pre}(x')$; 2) $|x'| < |x|$. Podem aplicar llavors l'H.I.

Correctesa d'un algorisme recursiu

- Si x no és un cas base ($|x| > 0$), apliquem l'hipòtesi d'inducció:

  H.I. = "Si x' compleix la precondició ($\texttt{Pre}(x')$ és cert) i $|x'| < |x|$ llavors l'algorisme acaba en temps finit i es compleix la postcondició ($\texttt{Post}(x')$)"

- Hem de demostrar que qualsevol crida recursiva $\texttt{F}(x')$ quan x no és un cas base ($|x| > 0$) compleix: 1) $\texttt{Pre}(x')$; 2) $|x'| < |x|$. Podem aplicar llavors l'H.I.

- Aplicant l'H.I. deduïm que després d'una crida recursiva $\texttt{Post}(x')$; cal demostrar que l'estat al qual s'arriba just després o fent alguns càlculs addicionals satisfà $\texttt{Post}(x)$

Potència ràpida

```
// Pre: x > 0 ∧ y ≥ 0
// Post: retorna x^y
int potencia(int x, int y);
```

Observem que

$$x^y = \begin{cases} 1 & \text{si } y = 0 \\ x \cdot (x^2)^\lambda & \text{si } y = 2\lambda + 1 > 0 \text{ és senar} \\ (x^2)^\lambda & \text{si } y = 2\lambda > 0 \text{ és parell} \end{cases}$$

Potència ràpida

```
int potencia(int x, int y) {
    if (y == 0) return 1;
    else if (y%2 == 1) return x*potencia(x*x,y/2);
    /* HI1: el resultat és (x^2)^{y/2} */
    else return potencia(x*x,y/2);
    /* HI2: el resultat és (x^2)^{y/2} */
}
```

- Acabament: podem agafar $|y| = y$, però també $|y| = \lceil 1 + \log_2(y) \rceil$, ja que $\lceil 1 + \log_2(y/2) \rceil = \lceil \log_2(y) \rceil < \lceil 1 + \log_2(y) \rceil$. Sempre enter (per això fem servir $\lceil \cdot \rceil$). Si $|y| \leq 0$ estem en un cas base ($\log_2 y \leq -1 \implies y \leq 1/2 \implies y \leq 0$). Si $|y| > 0$ llavors no estem en un cas base, $y \geq 1$ i $|y/2| < |y|$.

Potència ràpida

```
int potencia(int x, int y) {
    if (y == 0) return 1;
    else if (y%2 == 1) return x*potencia(x*x,y/2);
    else return potencia(x*x,y/2);
}
```

- Correcció: si $y = 0$ llavors retornem $x^0 = 1$. Si $y > 0$, es fa la crida recursiva `potencia(x*x,y/2)`. Com $x > 0$, tenim $x^2 > 0$. I com $y > 0$, llavors $y/2 \geq 0$. A més $|y/2| < |y|$. Es pot aplicar H.I.
- Correcció: si $y = 2\lambda$ és parell, per H.I. `potencia(x*x,y/2)` retorna $(x^2)^\lambda = x^{2\lambda} = x^y$ i la funció retorna el resultat correcte. Si $y = 2\lambda + 1$ és senar, per H.I. `potencia(x*x,y/2)` retorna $(x^2)^\lambda = x^{2\lambda} = x^{y-1}$; llavors la funció retorna $x \cdot x^{y-1} = x^y$, el resultat correcte.

Potència ràpida

Exercici: Demostreu la correcció de la següent implementació alternativa:

```
int potencia(int x, int y) {
    if (y == 0) return 1;
    else {
      int p = potencia(x, y/2);
      if (y%2 == 0) return p * p;
      else return x * p * p;
    }
}
```

Nombres binomials

```
// Pre: n ≥ m ≥ 0
// Post: retorna (n m)
int binomial(int n, int m);
```

Recordem:

$$\binom{n}{m} = \frac{n!}{m! \cdot (n-m)!}$$

és el nombre de subconjunts de $\{1, \ldots, n\}$ de mida m

Nombres binomials: Disseny

Hi ha diverses definicions recursives equivalents, que porten a solucions d'eficiència i elegància diferents
Triant $|(n,m)| = n$:

$$\binom{n}{m} = \begin{cases} 1 & \text{si } n = m \\ \frac{n\cdot(n-1)!}{m!\cdot(n-m)\cdot(n-1-m)!} = \frac{n}{n-m}\cdot\binom{n-1}{m} & \text{si } n > m \end{cases}$$

Nombres binomials

Triant $|(n,m)| = m$:

$$\binom{n}{m} = \begin{cases} 1 & \text{si } m = 0 \\ \frac{n! \cdot (n-m+1)}{m \cdot (m-1)! \cdot (n-m+1) \cdot (n-m)!} = \frac{n-m+1}{m}\binom{n}{m-1} & \text{si } m > 0 \end{cases}$$

Nombres binomials

Triant $|(n,m)| = m$:

$$\binom{n}{m} = \begin{cases} 1 & \text{si } m = 0 \\ \frac{n!\cdot(n-m+1)}{m\cdot(m-1)!\cdot(n-m+1)\cdot(n-m)!} = \frac{n-m+1}{m}\binom{n}{m-1} & \text{si } m > 0 \end{cases}$$

O bé descomposem així:

$$\binom{n}{m} = \begin{cases} 1 & \text{si } m = 0 \\ \frac{n\cdot(n-1)!}{m\cdot(m-1)!\cdot((n-1)-(m-1))!} = \frac{n}{m}\binom{n-1}{m-1} & \text{si } m > 0 \end{cases}$$

Nombres binomials

Triant $|(n,m)| = m$:

$$\binom{n}{m} = \begin{cases} 1 & \text{si } m = 0 \\ \frac{n!\cdot(n-m+1)}{m\cdot(m-1)!\cdot(n-m+1)\cdot(n-m)!} = \frac{n-m+1}{m}\binom{n}{m-1} & \text{si } m > 0 \end{cases}$$

O bé descomposem així:

$$\binom{n}{m} = \begin{cases} 1 & \text{si } m = 0 \\ \frac{n\cdot(n-1)!}{m\cdot(m-1)!\cdot((n-1)-(m-1))!} = \frac{n}{m}\binom{n-1}{m-1} & \text{si } m > 0 \end{cases}$$

(O bé fem servir el triangle de Tartaglia:

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$

però la solució és més ineficient)

Nombres binomials

```
// Pre: n ≥ m ≥ 0
// Post: retorna (n m)
int binomial (int n, int m) {
  if (m == 0) return 1;
  else return (binomial(n-1,m-1) * n) / m;
  /* HI: el resultat és (n-1 m-1) */
}
```

Observeu que $n \times \binom{n-1}{m-1}$ és sempre divisible entre m. Si hem de prendre cura dels overflows hauríem de fer primer la divisió de $\binom{n-1}{m-1}$ entre m i després fer el producte, però assegurant-nos abans que $\binom{n-1}{m-1} \geq m$.

Nombres binomials

- Acabament: podem agafar com a funció de mida simplement m.
- És evident que a cada crida recursiva decreix i arriba al cas base.
- l'H.I. és que el resultat de la crida recursiva és $\binom{n-1}{m-1}$
- Com que els paràmetres $n-1$ i $m-1$ compleixen la precondició i són més petits que n i m, es pot aplicar l'H.I.
- I per tant podem calcular $\binom{n}{m}$ a partir de $\binom{n-1}{m-1}$

Part I

[redacted]

Arbres binaris (o simplement arbres)

Un arbre o bé és l'arbre buit
o bé és un node anomenat arrel amb zero, u o dos
arbres successors anomenats fills o subarbres

Arbres binaris (o simplement arbres)

Un arbre o bé és l'arbre buit
o bé és un node anomenat arrel amb zero, u o dos
arbres successors anomenats fills o subarbres

Es presta a tractaments algorísmics recursius

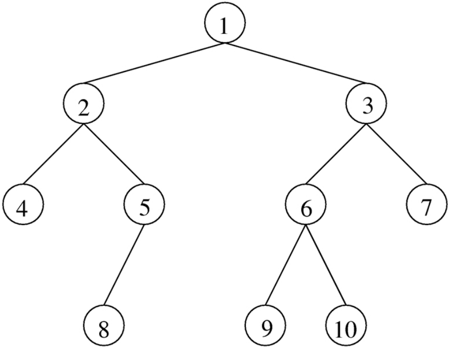
Arbres binaris (o simplement arbres)

Un arbre o bé és l'arbre buit
o bé és un node anomenat arrel amb zero, u o dos
arbres successors anomenats fills o subarbres

Es presta a tractaments algorísmics recursius

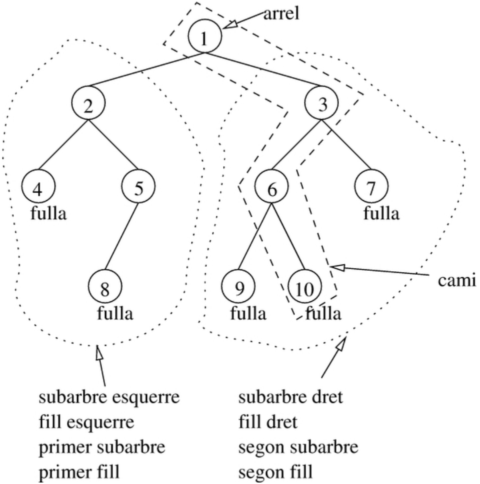
Els dos fills d'un node són anomenats esquerre i dret

Exemple d'arbre binari



No hem dibuixat els subarbres buits. La inclinació de cada aresta indica si el fill és dret o esquerre

Exemple d'arbre binari



Especificació dels arbres binaris

```
template <typename T> class BinTree {
public:
 BinTree();
 /* Pre: cert */
 /* Post: crea un arbre buit */

 BinTree(const T& x);
 /* Pre:  cert */
 /* Post: crea un arbre binari amb un sol node, l'arrel,
          que conté x, i els seus fills esquerre i dret
          són buits */

 BinTree (const T& x, const BinTree& left, const BinTree& right);
 /* Pre:  cert */
 /* Post: crea un arbre binari amb x a l'arrel,
          i left i right com com a fills esquerre
          i dret, respectivament */
```

Especificació dels arbres binaris

```
// Consultores:
bool empty() const;
/* Pre: cert */
/* Post: retorna cert si i només si
         l'arbre és buit */
BinTree left() const;
/* Pre: L'arbre implícit no és buit */
/* Post: retorna el fill esquerre de l'arbre implícit */

BinTree right() const;
/* Pre: L'arbre implícit no és buit */
/* Post: retorna el fill dret de l'arbre implícit */

const T& value() const;
/* Pre: L'arbre implícit no és buit */
/* Post: retorna el valor de l'arrel de l'arbre */
```

Especificació dels arbres binaris

- Cap modificadora! L'única manera de modificar un arbre és construir l'arbre modificat i assignar-lo a l'original.
- Totes les operacions que hem vist requereixen temps constant
- Important per al temps constant: tot és `const`, no es fan còpies dels fills

Part I

[redacted]

Mida d’un arbre

```
/* Pre: cert */
/* Post: El resultat és el nombre de nodes d’ a */
template <typename T>
int size(const BinTree<T>& a);
```

Mida d'un arbre

```
/* Pre: cert */
/* Post: El resultat és el nombre de nodes d'a */
template <typename T>
int size(const BinTree<T>& a) {
   if (a.empty()) return 0;
   else
     return 1 + size(a.left()) + size(a.right());
     /* HI1: el resultat és la mida d'a.left()*/
     /* HI2: el resultat és la mida d'a.right()*/
}
```

Mida d’un arbre

- La funció de mida és $|a|$.
- No confondre l’implementació de la funció de la mida d’un arbre amb la funció matemàtica mida.
- Qualsevol subarbre és més petit que l’arbre.
- L’H.I. s’aplica dos cops, una per cada crida.
- S’ha de comprovar que les dues crides compleixin la precondició.
- Tant `a.left()` com `a.right()` són més petits que `a`.
- Per tant a partir de la mida d’`a.left()` i d’`a.right()` podem calcular la mida d’`a`.

Alçària d’un arbre

Def.: L’alçària d’un arbre és la longitud del camí (nombre de nodes) més llarg de l’arrel a una fulla

Especificació:

```
/* Pre: cert */
/* Post: El resultat és l'alçària de l'arbre a*/
template <typename T>
int alcaria(const BinTree<T>& a);
```

Alçària d'un arbre

```
/* Pre: cert */
/* Post: El resultat és l'alçària de l'arbre a */
template <typename T>
int alcaria(const BinTree<T>& a) {
  if (a.empty())
    return 0;
  else
    return 1 + max(alcaria(a.left(),alcaria(a.right());
    /* HI1: el resultat és l'alçària d'a.left()*/
    /* HI2: el resultat és l'alçària d'a.right()*/
}
```

Alçària d'un arbre

- La funció de mida és $|a|$.
- Tant `a.left()` com `a.right()` són més petits que `a` i compleixen la precondició.
- L'H.I. aplicada a les dues crides recursives ens dona l'alçaria d'`a.left()` i d'`a.right()`.
- Triant el valor més alt dels dos i sumant u (per l'arrel) obtenim l'alçària d'`a` a partir de l'H.I.

Cerca d’un valor en un arbre

```
/* Pre: cert */
/* Post: El resultat indica si x és a l'arbre a o no */
template <typename T>
bool cerca(const BinTree<T>& a, const T& x);
```

Cerca d'un valor en un arbre

```
/* Pre: cert */
/* Post: El resultat indica si x és a l'arbre a o no */
template <typename T>
bool cerca(const BinTree<T>& a, const T& x) {
    bool b;
    if (a.empty()) b = false;
    else if (a.value() == x) b = true;
    else{
      b = cerca(a.left(),x);
      /* HI1: el resultat ens diu si x és a a.left() o no*/
      if (not b) b = cerca(a.right(),x);
      /* HI2: el resultat ens diu si x és a a.right() o no*/
    }
    return b;
```

És imprescindible que l'operador d'igualtat estigui definit per elements del tipus `T` `x == y` ha d'estar definit!

Cerca

- La funció de mida és $|a|$.
- Hi ha dos casos base, es poden calcular sense crida recursiva.
- Tant `a.left()` com `a.right()` són més petits que `a` i compleixen la precondició.
- L'H.I. aplicada a la primera crida recursiva ens diu si `x` és a `a.left()`. Si ja hi és no cal buscar més.
- Si no hi és, la segona crida recursiva ens diu si `x` és a `a.right()`.
- `x` no hi era ni a l'arrel ni al subarbre esquerra, per tant `x` és a `a` si i només si hi és a `a.right()`.

Sumar k als elements d’un arbre

```
/* Pre: cert */
/* Post: el resultat té la mateixa forma que a, el valor de
         cada node del resultat és la suma del valor del node
         corresponent d'a més k */
BinTree<int> suma(const BinTree<int>& a, int k);
```

Sumar k als elements d'un arbre

```
/* Pre: cert */
/* Post: el resultat té la mateixa forma que a, el valor de
         cada node del resultat és la suma del valor del node
         corresponent d'a més k */
BinTree<int> suma(const BinTree<int>& a, int k)
  if (a.empty())
    return BinTree<int>();
  else
    return BinTree<int>(a.value()+k, suma(a.left(), k),
                                      suma(a.right(),k));
    /*HI1: el resultat té la mateixa forma que a.left(), el valor
           de cada node del resultat és la suma del valor del node
           correspondent d'a.left() més k */
    /*HI2: el resultat té la mateixa forma que a.right(), el valor
           de cada node del resultat és la suma del valor del node
           correspondent d'a.right() més k */

}
```

Sumar k als elements d'un arbre

- Acabament: si $|a| = 0$ l'arbre és buit i estem en un cas base; amb $|a| > 0$ tenim un cas recursiu, i les crides a `suma` són amb $a.\mathtt{left}()$ i $a.\mathtt{right}()$ i $|a.\mathtt{left}()| < |a|$, $|a.\mathtt{right}()| < |a|$.
- Correcció: si $|a| = 0$ la solució retorna un arbre buit. Si $|a| > 0$ la precondició de les dos crides recursives es compleix i els paràmetres són de mida inferior: L'H.I. es pot aplicar.
- Correcció: A partir dels resultats de les crides recursives (per H.I.) i de l'arrel d'a a la qual s'ha sumat k podem construir el resultat desitjat per a.

Sumar k als elements d'un arbre

Fem-ho sobre el mateix arbre, com una acció:

```
/* Pre: a = A */
/* Post: deixa en a el resultat de sumar k a l'arbre A */
void suma(BinTree<int>& a, int k) {
  if (not a.empty()) { // si és buit no cal fer res
    BinTree<int> l = a.left();
    BinTree<int> r = a.right();
    suma(l, k);
    /* HI1: l és com a.left() amb k sumat a tots els nodes */
    suma(r, k);
    /* HI2: l és com a.right() amb k sumat a tots els nodes */
    a = BinTree<int>(a.value() + k, l, r);
  }
}
```

El raonament és similar al de l'anterior exemple.

Intersecció de dos arbres

```
/* Pre: cert */
/* Post: el resultat és l'arbre intersecció d'a i b */
BinTree<int> intersec(const BinTree<int>& a, const BinTree<int>& b)
{
  if (a.empty()) return BinTree<int>();
  else if (b.empty()) return BinTree<int>();
  else if (a.value() != b.value()) return BinTree<int>();
  else return BinTree<int>(a.value(), intersec(a.left(), b.left()),
                                      intersec(a.right(), b.right()));
  /* HI1: el resultat és la intersecció d'a.left() and b.left()*/
  /* HI2: el resultat és la intersecció d'a.right() and b.right()*/
  }
}
```

Intersecció de dos arbres

- Triem com a funció de mida $min(|a|, |b|)$
- Acabament: si $min(|a|, |b|) = 0$ un arbre al menys és buit i estem en un cas base; si $min(|a|, |b|) > 0$ tenim un cas base i un recursiu. Una crida a `intersec` és amb a.`left()` i b.`left()`; $min(|a.left()|, |b.left()|) < min(|a|, |b|)$. L'altra crida a `intersec` és amb a.`right()` i b.`right()`; $min(|a.right()|, |b.right()|) < min(|a|, |b|)$
- Correcció: si $min(|a|, |b|) = 0$ la solució retorna un arbre buit. si $min(|a|, |b|) > 0$ pel cas base la solució retorna un arbre buit. La precondició de les dues crides recursives es compleix i els paràmetres són de mida inferior: L'H.I. es pot aplicar.
- Correcció: Per H.I. a partir de la intersecció del dos arbres esquerres i la intersecció dels dos arbres drets podem construir l'intersecció de tots dos arbres amb l'arrel de a (o b).

Part I

[redacted]

Recorreguts d’arbres

Mètodes més habituals per visitar els nodes d’un arbre (per fer recorreguts o cerques):

- Recorreguts en profunditat

En tots els casos: recórrer l’arbre buit = no fer res

Recorreguts d'arbres

Mètodes més habituals per visitar els nodes d'un arbre (per fer recorreguts o cerques):

- Recorreguts en profunditat
  - En preordre

En tots els casos: recórrer l'arbre buit = no fer res

Recorreguts d’arbres

Mètodes més habituals per visitar els nodes d’un arbre (per fer recorreguts o cerques):

- Recorreguts en profunditat
  - En preordre
  - En inordre

En tots els casos: recórrer l’arbre buit = no fer res

Recorreguts d'arbres

Mètodes més habituals per visitar els nodes d'un arbre (per fer recorreguts o cerques):

- Recorreguts en profunditat
  - En preordre
  - En inordre
  - En postordre

En tots els casos: recórrer l'arbre buit = no fer res

Recorreguts d’arbres

Mètodes més habituals per visitar els nodes d’un arbre (per fer recorreguts o cerques):

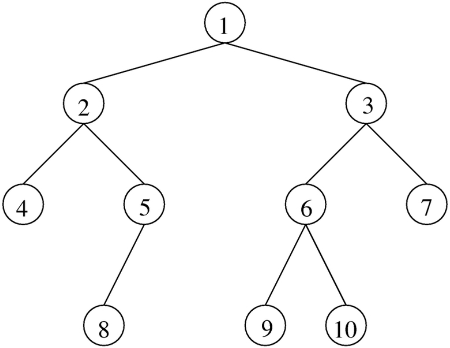
- Recorreguts en profunditat
  - En preordre
  - En inordre
  - En postordre
- Recorregut en amplada o per nivells

En tots els casos: recórrer l’arbre buit = no fer res

Recorreguts en profunditat: preordre

1. visitar l'arrel
2. recórrer fill esquerre (en preordre)
3. recórrer fill dret (en preordre)

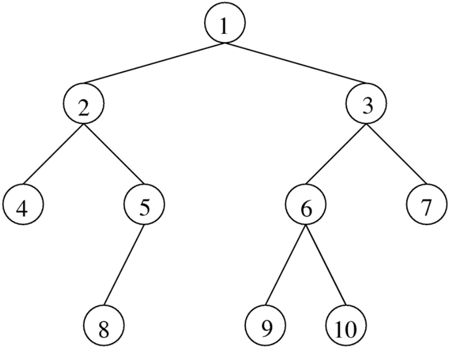
Exemple:
1, 2, 4, 5, 8, 3, 6, 9, 10 i 7



Recorreguts en profunditat: inordre

1. recórrer fill esquerre (en inordre)
2. visitar l'arrel
3. recórrer fill dret (en inordre)

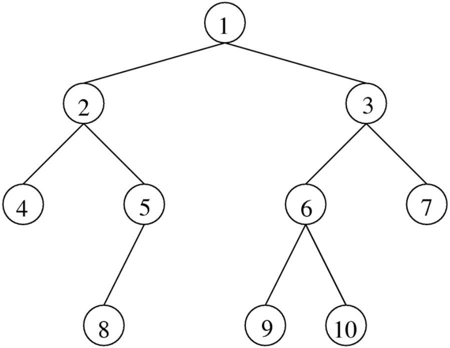
Exemple:
4, 2, 8, 5, 1, 9, 6, 10, 3, i 7



Recorreguts en profunditat: postordre

1. recórrer fill esquerre (en postordre)
2. recórrer fill dret (en postordre)
3. visitar l'arrel

Exemple:
4, 8, 5, 2, 9, 10, 6, 7, 3, i 1



Recorregut en preordre

```
/* Pre: cert */
/* Post: El resultat conté els nodes d'a en preordre */
template <typename T>
list<T> preorder(const BinTree<T>& a) {
    list<T> l;    // inicialment, buida
    if (not a.empty()) {
        l.insert(l.end(),a.value());
        l.splice(l.end(),preorder(a.left()));
        l.splice(l.end(),preorder(a.right()));
    }
    return l;
}
```

Recorregut en preordre

```
/* Pre: cert */
/* Post: El resultat conté els nodes d'a en preordre */
template <typename T>
list<T> preorder(const BinTree<T>& a) {
    list<T> l;    // inicialment, buida
    if (not a.empty()) {
        l.insert(l.end(),a.value());
        l.splice(l.end(),preorder(a.left()));
        l.splice(l.end(),preorder(a.right()));
    }
    return l;
}
```

Quina és la funció de mida? Quina l'H.I.? Com s'aplica? Penseu-hi!

Recorregut en preordre

```
/* Pre: cert */
/* Post: El resultat conté els nodes d'a en preordre */
template <typename T>
list<T> preorder(const BinTree<T>& a) {
    list<T> l;    // inicialment, buida
    if (not a.empty()) {
        l.insert(l.end(),a.value());
        l.splice(l.end(),preorder(a.left()));
        l.splice(l.end(),preorder(a.right()));
    }
    return l;
}
```

Quina és la funció de mida? Quina l'H.I.? Com s'aplica? Penseu-hi! Exercici: Com canviem les instruccions del mig per obtenir els recorreguts en inordre i en postordre?

Recorregut en inordre

Manera alternativa: afegir a una llista donada

Obtenim el recorregut fent una crida inicial amb la llista buida

```
/* Pre: l = L */
/* Post: l conté L seguida dels nodes d'a en inordre */
template <typename T>
void inorder(const BinTree<T>& a, list<T>& l) {
    if (not a.empty()) {
        inorder(a.left(),l);
        l.insert(l.end(),a.value());
        inorder(a.right(),l);
    }
}

// Ús:
BinTree<int> a;
...
list<int> rec;
inorder(a,rec);
```

Recorregut en amplada o per nivells

Visita d'els nodes d'un arbre donat de manera que:

- tots els nodes del nivell i s'han visitat abans que els del nivell $i + 1$
- dins de cada nivell, els nodes es visiten d'esquerra a dreta

Recorregut en amplada o per nivells

Es fa amb una cua

Repetir:

- agafar primer arbre de la cua;
- visitar la seva arrel;
- ficar els seus dos fills a la cua;

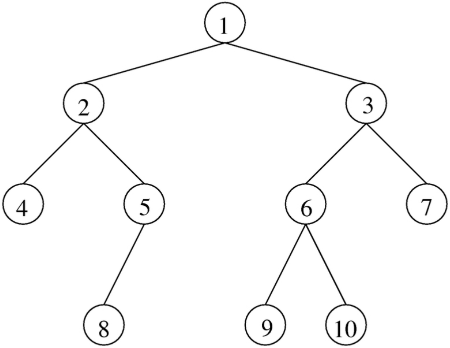
Recorregut en amplada o per nivells

Es fa amb una cua

Repetir:

- agafar primer arbre de la cua;
- visitar la seva arrel;
- ficar els seus dos fills a la cua;

Al llarg de tot l'algorisme, a cada iteració la cua conté alguns nodes del nivell k seguits de nodes del nivel $k+1$ que són fills dels nodes de nivell k que ja han sigut visitats i no són a la cua. En cap moment la cua conté nodes de més de dos nivells consecutius i mai un node de nivell $k+1$ precedeix un altre de nivel k a la c



Recorregut en amplada

```
/* Pre: cert */
/* Post: El resultat conté el recorregut d'a en amplada */
template <typename T>
list<T> nivells(const BinTree<T>& a) {
    list<T> l;  // inicialment, buida
    if (not a.empty()) {
        queue< BinTree<T> > c;
        c.push(a);
        while (not c.empty()) {
           BinTree<T> aux = c.front();
           c.pop();
           l.insert(l.end(),aux.value());
           if (not aux.left().empty()) c.push(aux.left());
           if (not aux.right().empty()) c.push(aux.right());
        }
    }
    return l;
}
```