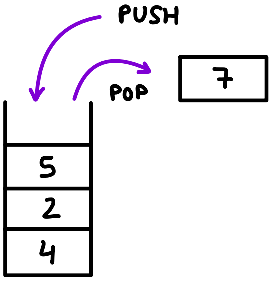
PILAS:

- **CONCEPTO:**



- **INCLUDE:** #include <stack> ...
- **DECLARACIÓN:** stack < T > p;
- **FUNCIONES:**
  - nombre. push (lo q metes)
  - nombre pop ()
  - bool que te dice si la pila esta vacia → nombre . empty ()
  - nombre . size ()
  - nombre . top () → primer elem de arriba

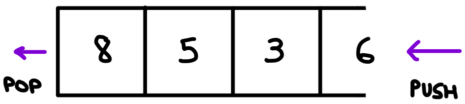
▷ LECTURA:

```
stack <int> p;
int n, x;
cin >> n;
for (int i = 0; i < n; ++i ) {
    cin >> x;
    p.push(x);
}
```

▷ ESCRITURA:

```
while (not p.empty( ) ) {
    cout << p.top << endl;
    p.pop( );
}
```

COLAS:



- **INCLUDE:** #include <queue> ...
- **DECLARACIÓN:** queue < T > p;
- **FUNCIONES:**
  - nombre. push (lo q metes)
  - nombre pop ()
  - bool que te dice si la pila esta vacia → nombre . empty ()
  - nombre . size ()
  - nombre . front () → primero del frente

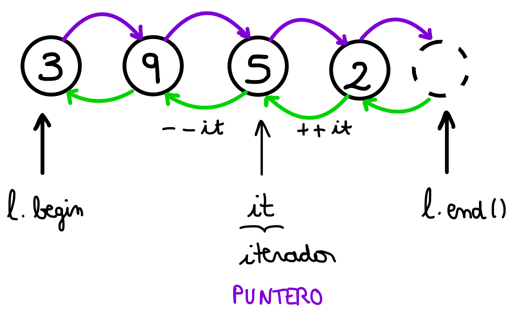
▷ LECTURA:

```
queue <int> q;
int n, x;
cin >> n;
for (int i = 0; i < n; ++i ) {
   cin >> x;
   q.push(x);
}
```

▷ ESCRITURA:

```
while (not q.empty ( ) ) {
    cout << q.front( ) << endl;
    q.pop( );
}
```

LISTAS:



ITERADORES:

- **DECLARACIÓN:**
  - LISTA CONSTANTE: list<T>::const_iterator it; (no modifican lista)
  - NO CONSTANTE: list<T>::iterator it;
- **DE CADA LISTA** se puede acceder al primer elem. (l.begin) y al de después del último (l.end).
- **CONSULTAR EL CONTENIDO DEL ELEM APUNTADO POR IT:** *it
- Avanzar y retroceder de 1 en 1 con ++it o --it.

LIST:

- `l1.splice(it, l2);` junta la lista `l2` a la `l1` donde apunta `it`, $l1 = [1,2,3,4,5,6], l2 = [10, 20, 50] \rightarrow$ `l1.splice(it, l2);` $\rightarrow l1 = [1,2,3,10,20,50,4,5,6], l2 = [\ ]$. Para concatenar las listas → `l1.splice(l1.end(), l2);`

- **INCLUDE:** #include <list>
- **DECLARACIÓN:** list <T> l;
- **FUNCIONES:**
  - l.insert (it, valor) → inserta valor delante de it
  - l.erase (it) → elimina el elem. apuntado por it, no retorna un iterador al elem sig del elim.
  - l.empty ()
  - l.size ()

LECTURA:

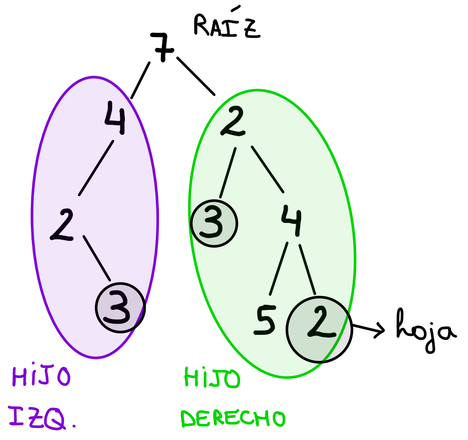
```
list <int> l;
int n, x;
cin >> n;
for (int i = 0; i < n; ++i ) {
    cin >> x;
    l.insert ( l.end( ), x );
}
```

ESCRITURA:

```
list <int> ::iterator it = l.begin( );
while (it != l.end( ) ) {
    cout << *it << endl;
    ++it;
}
```

ARBOLES:

CONCEPTO:



Podemos tener un arbol

- VACÍO
- HOJA (no hijo)
- 1 o 2 hijos

Dado un arbol solo podemos acceder a la raíz y a los arboles hijos.

⚠ **COMPILAR SIEMPRE CON:** -std=c++11

- Per nivells:

Es fa amb una cua:

1) Agafar primer arbre de la cua
2) Visitar la seva arrel
3) Ficar els seus dos fills a la cua que no són a la cua

Invariant: la cua conté alguns nodes del nivell k seguits dels fills dels nodes de nivell k

```
template <typename T>
list<T> nivells (const BinTree& a) {
    list<T> l;
    if (not a.empty()) {
        queue < BinTree<T> > c;
        c.push (a);
        while (not c.empty()) {
            Bintree<T> aux = c.front(); c.pop(); l.push_back (aux.value());
            if (not aux.left().empty()) c.push(aux.left());
            if (not aux.right().empty()) c.push (aux.right());
        }
    }
    return l;
}
```

BIN TREE (PROFES):

- **INCLUDE:** #include "BinTree.hh"
- **DECLARACIÓN:** BinTree <T> a;
- **FUNCIONES:**

BinTree<T>() ⟶ Construye un arbol vacío.

BinTree<T>(const T &x) ⟶ Construye un arbol solo con RAÍZ x

BinTree<T>(const T &x, const BinTree<T> &l, const BinTree<T> &r)

↳ Construye un arbol con RAÍZ, HIJO DERECHO y IZQUIERDO

T value() ⟶ Retorna el valor del arbol

BinTree<T> left() const ⟶ Retorna el subarbol izq. de este arbol (no vacío)

BinTree<T> right() const ⟶ Retorna el subarbol derecho de este arbol (no vacío)

bool empty() const ⟶ Nos dice si el arbol es vacío.

CORRECTESA DE PROGRAMES ITERATIUS

Correctesa de programes

Def: Per tots els valors inicials de les variables que satisfan la Pre, el programa acaba i els valors finals satisfan la Post.

Per demostrar que un programa és correcte:

- Raonament genèric → Inducció { Recursiu: Directament. / Iteratiu: Amagada en l'invariant.

Estats i assercions

Estat d'un programa: Tupla de valors de totes les variables.

Asserció: Descripció d'un conjunt d'estats.

↳ La Pre és l'asserció que se suposa que és certa al principi.

↳ La Post és l'asserció que volem que sigui certa al final.

Un programa és correcte si donada una Pre compleix la Post.

Correctesa dels programes iteratius (Exemples al PDF)

Invariant: Una asserció que és certa després de qualsevol nombre d'iteracions. Que una asserció INV sigui invariant es demostra per inducció.

Funció de fita: Variable / expressió que indica quantes iteracions queden.

Pasos:

0- Inventar una invariante I y una función cota f

Demostrar que:

```
// Pre: P
inicializaciones;
// Pre (del bucle): P'
while (B) {
cuerpo
}
// Post (del bucle) Q'
tratamiento final;
// Post: Q
```

1- Inicialización: Las inicializaciones del bucle establecen la invariante $P' \rightarrow I$ (Las variables de antes del bucle, diciendo el porqué).

2- Condición de salida: Caso en el que se cumple la invariante, pero no se cumple la condición de entrada del bucle, entonces, se cumple la postcondición

$I \wedge \neg B \Longrightarrow Q'$

3- Cuerpo del bucle: Si se cumple la invariante y se entra en el bucle, al final de una iteración vuelve a cumplirse la invariante: $/* I \wedge B */$ $cuerpo$ $/* I */$ Describe el bucle explicando los porqués de porque en la anterior iteración no se ha cumplido la postcondición y explica el caso en el que se cumple la postcondición y por qué se cumple también la invariante y porque no volvemos a entrar en el bucle.

4- Fin: La función de cota decrece en cada iteración: $/* I \wedge B \wedge f = F */$ $cuerpo$ $/* I \wedge f < F */$ Describe que el bucle es finito diciendo el porqué de ello (una variable crece/decrece).

Si entramos otra vez en el bucle, la función de la cota es estrictamente positiva:

$I \wedge B \Longrightarrow f > 0$

DISENY RECURSIU

- Recursió és inducció
- Funció d'immersió: Afegir més paràmetres
- Immersió d'eficiència: Afegir paràmetres per recordar càlculs.

Recursió, definicions recursives i inducció

Per aplicar recursivitat, primer cal trobar la definició recursiva del problema que se'ns demana.

Exemple: Suma elements d'una pila p:

$$\text{Suma}(p) = \begin{cases} 0 & \text{, si } p \text{ és buida} \\ p.top + suma(p.pop), & \text{altrament} \end{cases}$$

(El codi no accepta suma(p.pop()), es posa així per millor comprensió)

Principis de disseny recursiu

Cal identificar:
- Casos base, podem satisfer la Post amb càlculs directes
- Casos recursius, podriem satisfer la post amb paràmetres "més petits"

Cal demostrar: Amb tot valor x que satisfaci Pre(x), l'algorisme acaba (#finit de crides recursives) i acaba satisfent Post(x).

S'ha de demostrar que cada crida recursiva fa els paràmetres "més petits".

Esquema de correctesa d'un algorisme recursiu

- Demostrar que $\forall x$ tq Pre(x) cert $\Rightarrow$ es compleix Post(x)
  1) Si x cas base $\Rightarrow$ Demo directa
  2) Hipòtesi d'Inducció: $\forall x'$ tq Pre(x') cert i $x' < x$ $\Rightarrow$ es compleix Post(x')
  3) Si x cas recursiu $\Rightarrow$ comprovar que totes les crides a x' ($x' < x$) compleixen la Pre (x')
  4) Apliquem H.I. per veure que es compleix Post (x')
  5) Veiem que els càlculs ens porten a Post (x)
- Finalment, comprovem que totes les accions, funcions i mètodes cridats satisfan les precondicions respectives

3- Final: Tiene un número finito de llamadas recursivas. Decir que en la llamada recursiva se disminuye la cota (Ej: el árbol se hace más pequeño cuando se hace la llamada recursiva).

1- Caso sencillo: Demostración directa. Descripción de los casos que no se llaman a sí mismas

2- Caso recursivo: Aplicando H.I. deducimos que aplicando la llamada recursiva $Post(x')$ demostramos que el estado en el que llega después cumple $Post(x)$. Explica que, si no cumple el caso sencillo, entonces se puede seguir aplicando la llamada recursiva pero disminuyendo la búsqueda (Ej: en el árbol se llama a los sub-árboles) y se concluye diciendo que la postcondición se obtiene por hipótesis de inducción con llamadas recursivas.

Inmersión de funciones: Se crea una función de inmersión cuando la función original no tiene parámetros suficientes. Cuando la función original llama a la función de inmersión (función auxiliar), se añaden algunos parámetros adicionales ignorando algunos de los resultados devueltos.

- Debilitamiento del post: la llamada recursiva solo hace una parte del trabajo. (el return del caso base es un número).
- Reforzamiento del pre: la llamada recursiva recibe una parte del trabajo ya hecho, y esta la completa. (return del caso base es una variable).

Eliminación de cálculos repetidos:

Iteración:

- Añadir variables locales que recuerden cálculos ya hechos para la siguiente iteración.
- No aparecen en la Pre ni en la Post. La especificación no cambia.
- Pero aparecen en la invariante. Hay que decir que vale cada iteración.

Recursión:

- Las variables locales no sirven (se crean nuevas en cada iteración).
- Función de inmersión de eficiencia recursiva: nuevos parámetros de entrada o salida.
- Se tienen que añadir en la Pre/Post.
- La función original no es recursiva, llama a la de inmersión.

Inmersiones de eficiencia: introducir parámetros o resultados adicionales para transmitir valores ya calculados en/a otras llamadas.

Eficiencia y consideraciones generales: Las funciones deben ser eficientes en tiempo y en memoria. (Ej: Con un vector de tamaño n, tiempo proporcional a seq(n))