- Contendor: almacena objetos.
- Iteradores: apuntador que referencia la posición de memoria de un elemento de una lista.
    - Iterador con el cual se puede modificar el dato.

```
lis<OBJETO>::iterator it = l.begin();
lis<OBJETO>::iterator it = l.end();
```

    - Iterador de solo consulta

```
list<OBJETO>::const_iterator it = l.begin();
```

    - Si `list<int> l` es una lista vacía `l.begin() = l.end()`.
    - `l.end()` no contiene ningún objeto, simplemente indica el final de la lista.
    - El iterador se incrementa/decrementa haciendo `it++/it--` únicamente.
    - Para acceder al apuntador se usa (`*it`).
- `it = l.erase(it);` Borra el elemento al que apunta `it` y `it` pasa al elemento siguiente.
- `it = l.insert(it, 5);` introduce el elemento a la posición dónde apunta `it`, pero `it` sigue apuntando al mismo elemento $l = [1, \mathbf{3}, 4,5,8] \rightarrow$ `l.insert(it, 5);` $\rightarrow l = [1,5, \mathbf{3}, 4,5,8]$.
- `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);`

Constructoras:

BinTree<OBJETO> a; //Crea un BinTree vacío

BinTree<OBJETO> a(s); //Crea un BinTree de nodo s y sin hojas

BinTree<OBJETO> a(s, b1, b2); //Crea un BinTree con nodo s y con sub-árbol izquierdo b1 y derecho b2

Consultoras:

a.empty(); //Devuelve un bool de si el árbol es vacío

a.left(); //Devuelve la sub-árbol izquierdo

a.right(); //Devuelve la sub-árbol derecho

a.value(); //Devuelve el objeto del nodo

Recorridos de árboles

En profundidad:

Preorden	Inorden	Postorden
1- Visitar raíz 2- Recorrer sub-árbol izquierdo (en preorden) 3- Recorrer sub-árbol derecho (en preorden)	1- Recorrer sub-árbol izquierdo (en Inorden) 2- Visitar raíz 3- Recorrer sub-árbol derecho (en Inorden)	1- Recorrer sub-árbol izquierdo (en postorden) 2- Recorrer sub-árbol derecho (en postorden) 3- Visitar raíz

En amplitud o por niveles:

Se hace una cola:

1- Coger el primer árbol de la cola
2- Visitar su raíz
3- Meter sus dos sub-árboles en la cola

Corrección de un programa: Si el estado inicial de un programa o función cumple la precondición, entonces el programa acaba en un nombre finito de pasos y el estado final cumple la postcondición.

Estado de un programa = valores de todas las variables

Aserción: Descripción del conjunto de estados.

- Precondición: es la aserción que el estado inicial debe satisfacer.
- Postcondición: es la aserción que debe ser cierta para el estado final, en caso contrario no satisface la especificación.

Corrección de programas iterativos:

- Invariante (I): Una aserción que es cierta después de cualquier número de iteraciones (0 incluido). Que una aserción invariante es invariante se demuestra por inducción.
- Función cota: Función sobre las variables que dice cuántas iteraciones quedan como mucho.

Pasos:

0- Inventar una invariante I y una función cota f

Demostrar que:

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:

$I \wedge B \Longrightarrow f > 0$

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

Para aplicar recursividad, se necesita aplicar una definición recursiva, para encontrarla consideramos operaciones que descomponen un dato en elementos más pequeños. Si hay que formalizarla utilizamos inducción. Ejemplo de suma de elementos de una pila inductivamente:

$$suma(P) = \begin{cases} 0 & Si\ P\ está\ vacía \\ cima(P) + suma\big(desapilar(P)\big) & Si\ P\ no\ está\ vacía \end{cases}$$

Recursivamente de búsqueda de una pila:

- Si p está vacía, x no aparece a p
- Si P no está vacía $x \in p \Leftrightarrow x = cima(P) \vee x \in desapilar(p)$

Principios de diseño recursivo:

- Caso/s base: valores de parámetros en los cuales podemos cumplir la postcondición con cálculos directos.
- Caso/s recursivos: valores de parámetros en los cuales podemos cumplir la postcondición si tuviéramos el resultado por algunos parámetros "más pequeños".

Corrección de un algoritmo recursivo:

Demostrar que $\forall x\ tq\ cumpla\ Pre(x) \implies se\ cumple\ Post(x)$, en un número finito de llamadas recursivas.

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

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).

Pro 2 – Mejoras de eficiencia

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))