



ALGORITMOS Y ESTRUCTURAS DE DATOS

DISEÑO Y ANÁLISIS DE ALGORITMOS

ALGORITMO: Secuencia de instrucciones, cada una de las cuales representa una tarea bien definida y puede ser llevada a cabo en una cantidad finita de tiempo y con un número finito de recursos computacionales.

Un requerimiento fundamental es que el algoritmo *debe terminar en un número finito de pasos*, de esta manera él mismo *puede ser usado como una instrucción en otro algoritmo más complejo*.

En otras palabras, es un procedimiento (conjunto de instrucciones bien definidas) para cumplir una tarea. A partir de un **estado inicial**, el algoritmo debe terminar en un **estado final** bien definido.

El algoritmo a desarrollar debe **particionar** las tareas en una serie de p **etapas**. Las etapas son simplemente subconjuntos de las tareas y, para ser una **partición admisible**, debe satisfacer las siguientes restricciones:

- Cada tarea debe estar en una y sólo una etapa. De lo contrario, la tarea no se realizaría o se realizaría más de una vez. Siguiendo esta idea, se busca partitionar el conjunto de tareas en subconjuntos disjuntos.
- Las tareas a ejecutarse en una dada etapa no deben acceder al mismo objeto.

ESTRUCTURAS

La implementación de los algoritmos se hace usando estructuras, tenemos las más simples (escalares, arreglos y matrices) y otras más complejas (listas, pilas, colas, árboles, grafos, conjuntos) que tienen ciertas ventajas:

- Se ahorra tiempo en programación ya que no es necesario codificar.
- Estas implementaciones suelen ser eficientes y robustas.
- Se separan dos capas de código bien diferentes, por una parte, el algoritmo que escribe el programador y por otro las rutinas de acceso a las diferentes estructuras.
- Existen estimaciones bastante uniformes de los tiempos de ejecución de las diferentes operaciones.
- Las funciones asociadas a cada estructura son relativamente independientes del lenguaje o la implementación en particular. Una vez que se plantea un algoritmo en términos de operaciones sobre una tal estructura es fácil implementarlo en una variedad de lenguajes con una performance similar.

La **eficiencia** de un código va en forma inversa con la cantidad de recursos que consume, principalmente **tiempo de CPU** y **memoria**. A veces en programación la eficiencia se contrapone con la sencillez y legibilidad del código.

Es importante saber cuándo y dónde preocuparse por la eficiencia. En algunos componentes del programa no es necesario preocuparse por la eficiencia (por ejemplo, cuando representan un 5% del tiempo total del cálculo, donde es preferible preocuparse por la robustez y sencillez de programación).

ESTRATEGIAS

- **Búsqueda exhaustiva:** Busco todos los caminos y me quedo con el menor. Si bien nos da la solución óptima, no es para nada práctico ya que suele implicar una cantidad excesiva de operaciones: mientras más vértices a tomar en cuenta, el costo computacional crece de tal manera que deja de ser aplicable a partir de una cantidad relativamente pequeña.
- **Estrategia heurística:** Es un costo menor pero no se obtiene la solución óptima generalmente. Se basa en encontrar una solución relativamente buena (que puede llegar a ser la solución óptima pero también depende de cómo se ejecute el algoritmo).

Un “**Tipo Abstracto de Datos**” (TAD) es la descripción matemática de un objeto abstracto, definido por las operaciones que actúan sobre el mismo.

Cuando usamos una estructura compleja como un conjunto, lista o pila podemos separar tres niveles de abstracción diferente (de mayor a menor abstracción): las operaciones abstractas sobre el TAD, la interfaz concreta de una implementación y la implementación de esa interfaz.

El **TAD CONJUNTO** tiene como características y operaciones abstractas:

- Contiene elementos, los cuales deben ser diferentes entre sí.
- No existe un orden particular entre los elementos del conjunto.
- Se pueden insertar o eliminar elementos del mismo.
- Dado un elemento se puede preguntar si está dentro del conjunto o no.
- Se pueden hacer las operaciones binarias bien conocidas entre conjuntos a saber, intersección y diferencia.

La interfaz es el conjunto de operaciones (con una sintaxis definida) que producen las operaciones del TAD. La implementación de estas funciones es el código específico que implementa cada una de las funciones declaradas en la interfaz.

El tiempo de ejecución de un programa depende de:

1. La eficiencia del compilador y las opciones de optimización que pasamos al mismo en tiempo de compilación.
2. El tipo de instrucciones y la velocidad del procesador donde se ejecutan las instrucciones compiladas.
3. Los datos del programa (la cantidad de números o la distribución estadística: si son todos iguales o están ordenados/casi ordenados).
4. La complejidad algorítmica del algoritmo subyacente.

- **Suma de arreglo de n números:** Depende de la longitud n del arreglo tal que $T(n) = cn$, donde c es una constante que representa el tiempo necesario para sumar un elemento.
- **Búsqueda de la ubicación de un elemento l en un arreglo a:** Depende de la ubicación del elemento dentro del arreglo de tamaño n, tal que:

$$\begin{aligned} T(n) &= cj & T_{\text{promedio}}(n) &= cn/2 \\ T_{\text{peor}}(n) &= cn & T_{\text{mejor}}(n) &= c \end{aligned}$$

NOTACIÓN ASINTÓTICA

T(n) = O(f(n)) se lee “T(n) es orden f(n)” y se cumple cuando existen constantes $c, n_0 > 0$ tales que $T(n) \leq c f(n)$, para $n \geq n_0$.

Esta notación se utiliza para el número de operaciones necesarios para ejecutar un cierto algoritmo (relacionado al tiempo de cómputo), pero se puede aplicar para cualquier recurso como, por ejemplo: memoria total requerida (o sea, un orden para la memoria adicional), cantidad de sockets que tengo que abrir para comunicarme con el exterior.

- No nos interesa cómo se comporta la función T(n) para valores de n pequeños, sino sólo la tendencia para $n \rightarrow \infty$.
- Un tiempo “**polinomial**” (P) es aquél tal que $T(n) = O(n^a)$ para algún a. Aquellos algoritmos que tienen tiempo de ejecución mayor que cualquier polinomio (funciones exponenciales $a^n, n!, n^n$) se les llama “no polinomiales”.
- Llamamos a $f(n)$ la “**tasa de crecimiento**” de T(n) y generalmente buscamos la cota superior menor o inferior mayor. Las funciones más típicas utilizadas como tasa de crecimiento son:

$1 < \log n < n^{1/2} < n < n^2 < \dots < n^p < 2^n < 3^n < \dots < n! < n^n$, donde el operador < representa O(...) tal que $1 = O(\log n)$, $\log n = O(n^{1/2})$.

• PROPIEDADES:

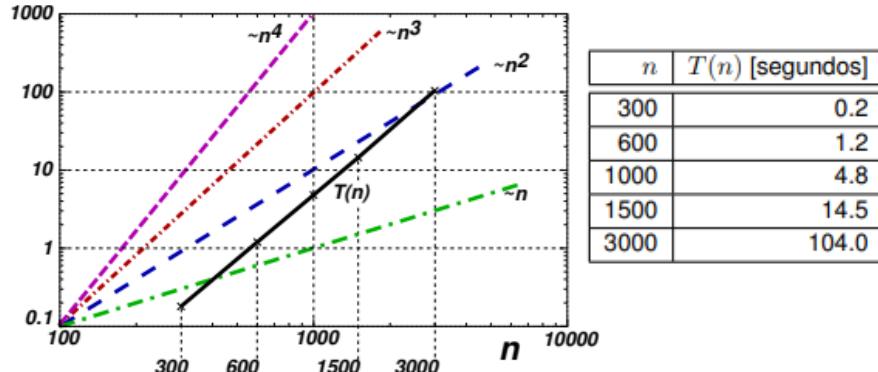
- **Invariancia ante constantes multiplicativas:** $T(n) = O(c f(n))$ es equivalente a $T(n) = O(f(n))$.
Demostración: Por ejemplo, para $T(n) = O(2n^3)$, existen c y n_0 tales que $T(n) \leq c2n^3$ para $n \geq n_0$, pero entonces también podemos decir que $T(n) \leq c' n^3$ para $n \geq n_0$ con $c' = 2c$.
- **Invariancia de la tasa de crecimiento ante valores en un conjunto finito de puntos:** Si $T_1(n) = 100$ para $n < 10$ y $T_1(n) = (n+1)^2$ para $n \geq 10$, entonces $T_1(n)$ coincide con la función $T(n) = (n+1)^2$.
Demostración: Si $T(n) \leq 2n^2$ para $n \geq 3$, entonces $T_1(n) < 2n^2$ para $n > n'_0 = 10$.
- **Transitividad:** Si $T(n) = O(f(n))$ y $f(n) = O(g(n))$ entonces $T(n) = O(g(n))$.
Demostración: Si $T(n) \leq cf(n)$ para $n \geq n_0$ y $f(n) \leq c' g(n)$ para $n \geq n'_0$. Entonces $T(n) \leq c'' g(n)$ para $n \geq n''_0$, donde $c'' = cc'$ y $n''_0 = \max(n_0, n'_0)$.
- **Equivalencia:** Si dos funciones f y g satisfacen que $f(n) = O(g(n))$ y $g(n) = O(f(n))$ entonces decimos → $n! \sim \sqrt{2\pi} n^{n+1/2} e^{-n}$ que sus tasas de crecimiento son equivalentes y se denota por $f \sim g$.

DETERMINACIÓN EXPERIMENTAL DE LA TASA DE CRECIMIENTO

Se corre el programa para una serie de valores de n, se toman los tiempos de ejecución y a partir de estos datos se obtiene la tasa de crecimiento.

Se grafican los valores encontrados (tabla) en ejes logarítmicos (es decir, graficar $\log T(n)$ en función de $\log n$) para así obtener líneas rectas. Luego, se grafican varias funciones n^a y se busca una que sea paralela a $T(n)$: en el caso del ejemplo es n^3 .

Cuando el tiempo de ejecución es exponencial, es decir $\sim a^n$ se prefieren ejes semi-logarítmicos ($\log T(n)$ en función de n).



Si no sabemos el tipo de crecimiento, se procede en forma incremental. Se prueba un gráfico logarítmico, si se tiene una recta como curva, entonces es una potencia. Si en vez de ser una recta es una curva que tiende a acelerarse cada vez más, probamos con las exponentiales. Si crece todavía más que las exponentiales, puede ser del tipo $n!$ o n^n .

ALGORITMOS P Y NP

TIEMPOS DE EJECUCIÓN: Instrucciones realizadas en una máquina de Turing.

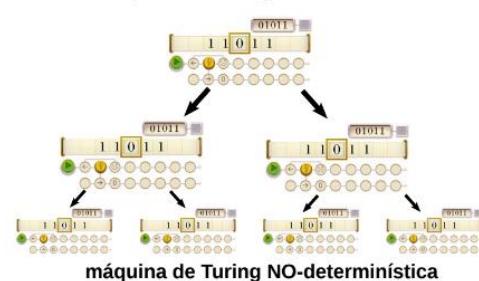
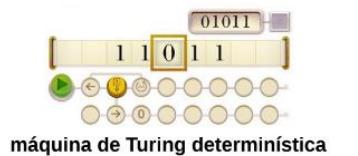
MÁQUINA DE TURING: Abstracción de la computadora más simple posible, con un juego de instrucciones reducido. Una máquina de Turing no determinística (no existe en la práctica, es un concepto imaginario) es una máquina de Turing que en cada paso puede invocar un cierto número de instrucciones, y no una sola instrucción como es el caso de la máquina de Turing determinística. Esto es, es una máquina de Turing que tiene ramificaciones de máquinas de Turing.

• CLASE DE PROBLEMAS DE COMPLEJIDAD NP (NON-DETERMINISTIC POLYNOMIAL):

- (1) Un problema es NP si tiene un tiempo de ejecución polinomial en una máquina de Turing no determinística. Todos los problemas, prácticamente.
- (2) Cada instancia del problema tiene un tiempo polinomial, luego el crecimiento extraordinario del costo computacional se produce por la enorme cantidad de instancias que hay.

• CLASE DE PROBLEMAS DE COMPLEJIDAD P (POLYNOMIAL):

Un problema es P si existe algún algoritmo polinomial (sort(vector), sum(vector)).



• PROBLEMAS NP-COMPLETOS (NPC):

Un problema es NPC si cualquier problema de NP se puede reducir a ese problema. Son los candidatos a tener la más alta complejidad algorítmica de NP ya que, si un problema A se puede resolver en términos de otro problema B (esto es, se reduce a B), A tiene menor o igual complejidad algorítmica que B.

• ¿P = NP? DILEMA MILLONARIO:

Si se puede demostrar que algún problema de NPC tiene complejidad algorítmica no-polinomial (y por lo tanto, todos los problemas de NPC), entonces $P \neq NP$.

Si se encuentra algún algoritmo de tiempo polinomial para un problema de NPC, entonces todos los problemas de NPC (y por lo tanto de P) serán P, es decir, $P = NP$.

No siempre hay un solo parámetro n que determina el tamaño del problema. Para los grafos, el tiempo de ejecución depende del número de vértices m y el número de aristas n_e : $T(m, n_e)$. Luego, es conveniente fijar un parámetro o definir un parámetro adimensional como la "tasa de ralitud":

$$s = \frac{n_e}{m(m-1)/2}$$

CONTEO DE OPERACIONES PARA EL CÁLCULO DEL TIEMPO DE EJECUCIÓN

Asumiendo que no hay llamadas recursivas en el programa, la regla básica para calcular el tiempo de ejecución de un programa es **ir desde los lazos o construcciones más internas hacia las más externas**. Se comienza asignando un costo computacional a las sentencias básicas. A partir de esto se puede calcular el costo de un bloque, sumando los tiempos de cada sentencia. Lo mismo se aplica para funciones y otras construcciones sintácticas tales como:

(1) Bloques if: Se puede considerar el peor caso o el caso promedio, calculando la probabilidad P de que `<cond>` sea verdadero.

```
if (cond) {
    <body>
}
Tpeor = Tcond + Tbody
Tprom = Tcond + PTbody
```

En el caso de que tenga un bloque else, entonces:

$$T_{\text{peor}} = T_{\text{cond}} + \max(T_{\text{body-true}}, T_{\text{body-false}}) \leq T_{\text{cond}} + T_{\text{body-true}} + T_{\text{body-false}}$$

$$T_{\text{prom}} = T_{\text{cond}} + PT_{\text{body-true}} + (1-P) T_{\text{body-false}}$$

(2) Lazos: El caso más simple es cuando el lazo se ejecuta un número fijo de veces y el cuerpo del lazo tiene un tiempo de ejecución constante.

```
for (i = 0; i < N; i++) {
    <body>
}
donde Tbody = constante:
T = Tini + N(Tbody + Tinc + Tstop)
```

T_{ini} = tiempo de ejecución de la parte de "inicialización" del lazo: $i = 0$.
 T_{inc} = tiempo de ejecución de la parte de "incremento" del contador: $i++$.
 T_{stop} = tiempo de ejecución de la parte de "detención" del contador: $i < N$.

Cuando T_{body} no es constante, debemos evaluar explícitamente la suma de todas las contribuciones:

$$T = T_{\text{ini}} + \sum_{i=0}^{N-1} (T_{\text{body},i} + T_{\text{inc}} + T_{\text{stop}}).$$

Cuando no se puede calcular una expresión analítica para las sumas, si es posible determinar una cierta tasa de crecimiento para todos los términos:

$$T_{\text{body},i} + T_{\text{inc}} + T_{\text{stop}} = O(f(n)), \quad \text{para todo } i$$

$$T \leq N \max_{i=1}^{N-1} (T_{\text{body},i} + T_{\text{inc}} + T_{\text{stop}}) = O(Nf(n))$$

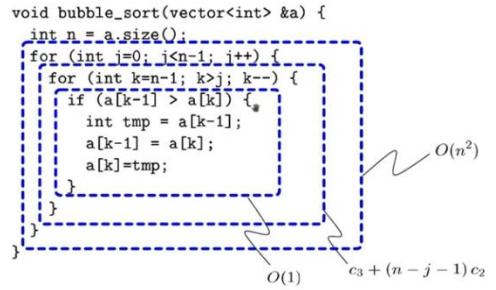
Más difícil aún es el caso en que el número de veces que se ejecuta el lazo no se conoce a priori, por ejemplo, un while, ya que se debe determinar también el número de veces que se ejecutará el lazo.

IMPORTANTE PARA BUCLES:

Si tengo un rango $[a, b]$, tengo $b - a + 1$ enteros. Para $\text{for}(i=n-1; i>j, i-)$, el rango es $[j+1, n-1]$ entonces tengo $n-j+1$ enteros.

BUBBLE-SORT:

Para $j = 0$, el menor elemento de todos es insertado en $v[0]$ mediante una serie de intercambios. A partir de ahí, $v[0]$ no se toca más y para $j = 1$ el mismo procedimiento es aplicado al rango de índices que va desde $j = 1$ hasta $j = n-1$, hasta ordenar todo el vector.



- El cuerpo (if) es de orden de 1 porque no crece con el tamaño del vector. Es constante.
- El bucle interno se ejecuta $n - j - 1$ veces, se puede descomponer tal que $T = c_3 + (n - j - 1)c_2$, donde c_3 es T_{ini} y $c_2 = T_{\text{inc}} + T_{\text{stop}} + T_{\text{body}}$
- El bucle externo se ejecuta $n - 1$ veces. Luego, no puedo usar el criterio de $T = T_{\text{ini}} + N(T_{\text{body}} + T_{\text{inc}} + T_{\text{stop}})$ porque lo que está adentro es variable. Luego:

```
Tini=c4    Tstop=c5    Tinc=c6
for (int j=0; j<n-1; j++) {
    for (int k=n-1; k>j; k--) {
        if (a[k-1] > a[k]) {
            int tmp = a[k-1];
            a[k-1] = a[k];
            a[k]=tmp;
        }
    }
}
Tbody=c3 +(n-j-1) c2
```

Utilizo la fórmula con la suma explícita de todas las contribuciones. Vemos que c_5, c_6, c_3 son constantes, luego puedo sacarlos afuera por la cantidad de veces que se ejecutan (rango: $[0, n-2]$, luego $(n-1)$). De nuevo puedo sacar la constante c_2 fuera, ahora multiplicando a la suma.

$$T = c_4 + (n-1)(c_3 + c_5 + c_6) + c_2 \sum_{j=0}^{n-2} (n-j-1)$$

$$\sum_{j=1}^n j \approx \int_0^n j \, dj = \frac{n^2}{2} = O(n^2).$$

Técnica suma

Encuentro la solución para la suma:

$$\sum_{j=0}^{n-2} (n-j-1) = (n-1) + (n-2) + \dots + 1 = \left(\sum_{j=1}^n j \right) - n,$$

$$\sum_{j=1}^n j = \frac{n^2}{2} + \frac{n}{2} = \frac{n(n+1)}{2}$$

$$T(n) = \underbrace{c_4}_{O(1)} + \underbrace{(n-1)(c_3 + c_5 + c_6)}_{\overbrace{\quad}^{O(n)}} + \underbrace{c_2 \frac{n(n+1)}{2}}_{\overbrace{\quad}^{O(n^2)}}$$

$$= O(\frac{n^2}{2}) \quad \text{También } O(n^2)$$

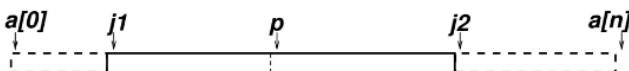
TIEMPOS DE EJECUCIÓN: LLAMADAS A FUNCIONES

Si hacemos un grafo del algoritmo y este no tiene ciclos, es porque no tiene recursividad. Si no hay recursividad, se puede aplicar un cierto principio para las llamadas a rutinas:

Una vez calculados los tiempos de ejecución de rutinas que no llaman a otras rutinas (las del conjunto S_0 en el gráfico), calculamos el tiempo de ejecución de aquellas rutinas que sólo llaman a las rutinas de S_0 (conjunto S_1). Así sucesivamente hasta llegar al main.

Se asigna a las líneas con llamadas a funciones de S_0 de acuerdo con el tiempo de ejecución previamente calculado, como si fuera una instrucción más del lenguaje. Por ejemplo, podemos usar a bubble como una instrucción cuyo costo es $O(n^2)$.

Para llamadas recursivas, no sirve este principio. Se hace uso de la **búsqueda binaria**, esto es, llegar a una expresión recursiva para el tiempo de ejecución mismo. La precondition para esto es que el vector esté ordenado de menor a mayor, sino da cualquier cosa.



```

1. int bsearch2(vector<int> &a, int k, int j1, int j2) {
2.     if (j1==j2-1) {
3.         if (a[j1]==k) return j1;
4.         else return j2;
5.     } else {
6.         int p = (j1+j2)/2;
7.         if (k<a[p]) return bsearch2(a,k,j1,p);
8.         else return bsearch2(a,k,p,j2);
9.     }
10. }
11. int bsearch(vector<int> &a, int k) {
12.     int n = a.size();
13.     if (k<a[0]) return 0;
14.     else return bsearch2(a,k,0,n);
15. }
16. }
```

lower_bound: dado un vector de una serie de elementos ordenados y otro elemento k que busco en ese vector, encuentro la primera posición donde podría insertar al elemento para que siga estando ordenado.

Para $n = 2^p$:

Como la función es recursiva, la relación para su tiempo de ejecución es recursiva también.

El $T(n/2)$ viene del bsearch2, en el else donde saca la mitad.

Se ve que el número que acompaña a la d es el mismo q el exponente.

$$T(n) = \begin{cases} c & ; \text{ si } n = 1; \\ d + T(n/2) & ; \text{ si } n > 1; \end{cases}$$

$$T(2) = d + T(1) = d + c$$

$$T(4) = d + T(2) = 2d + c$$

$$T(8) = d + T(4) = 3d + c$$

⋮

$$T(2^p) = d + T(2^{p-1}) = pd + c$$

como $p = \log_2 n$, vemos que

$$T(n) = d \log_2 n + c = O(\log_2 n)$$

Las listas se representan por celdas encadenadas por punteros, entonces se representan a las **posiciones como punteros a las celdas**, de manera que las posiciones se asuman como objetos abstractos y no como enteros. Al no ser enteros y no necesariamente ser válidas para hacer operaciones de enteros, se definen operaciones abstractas.

Se define: `list<int>::iterator it = L.begin();` ó `auto it = L.begin();`



$$T(n) = \begin{cases} c & ; \text{ si } n = 1; \\ d + 2T(n/2) & ; \text{ si } n > 1; \end{cases}$$

$$T(2) = d + 2T(1) = d + 2c$$

$$T(4) = d + 2T(2) = d + 2d + 4c = 3d + 4c$$

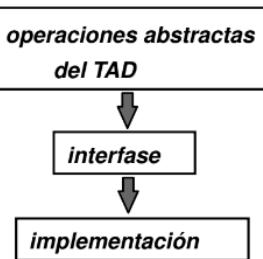
$$T(8) = d + 2T(4) = d + 6d + 8c = 7d + 8c$$

⋮

$$T(n) = (n - 1)d + nc \leq (c + d)n = O(n), \quad n = 2^p$$

En vez de acotar el tiempo del if usando max como arriba, si tomamos la suma llegamos a la estimación:
Es correcto también pero el máximo es una mejor estimación.

TIPOS DE DATOS ABSTRACTOS FUNDAMENTALES: TAD LISTA



- **TAD LISTA:** Una lista es una secuencia de cero o más elementos de un tipo determinado elem_t (por ejemplo int o double): $L = (a_0, a_1, \dots, a_{n-1})$ es una lista de n elementos tal que $n \geq 0$. Si $n = 0$, es decir, la longitud de la lista es cero, la lista está vacía.

Existe un orden lineal entre los elementos de la lista, tal que $(1,3,7,4) \neq (3,7,4,1)$. Se dice a_i está en la posición i . Introducimos una posición ficticia n que está fuera de la lista, la cual va cambiando a medida que se insertan o eliminan elementos de la lista. Luego, es conveniente el método end() además del begin().

A las posiciones en el rango $[0, n-1]$ las llamamos **dereferenciables** ya que pertenecen a un objeto real y, por lo tanto, podemos obtener una referencia a ese objeto.

- OPERACIONES ABSTRACTAS:

1. **erase:** suprime el elemento que se encuentra en una posición p dada. Sólo es válido suprimir en las posiciones dereferenciables.
2. **insert:** inserta el elemento x en una posición dada. Los elementos que le siguen al insertado se desplazan una posición. Es posible insertar en cualquier posición dereferenciable y también en la posición ficticia end().
3. Acceder al elemento en la posición p ($a_p <- x$), tanto para modificar al valor como para acceder a él ($x <- a_p$)
4. Avanzar una posición, es decir, dada una posición p correspondiente al elemento a_i retornar la posición q correspondiente al elemento a_{i+1} .
5. Retornar la primera posición de la lista y la posición ficticia al final de la lista.

- INTERFAZ SIMPLE PARA LISTAS:

```

1. class iterator_t { /* . . . */ };
2.
3. class list {
4. private:
5. // ...
6. public:
7. // ...
8. iterator_t insert(iterator_t p, elem_t x);
9. iterator_t erase(iterator_t p);
10. elem_t & retrieve(iterator_t p);
11. iterator_t next(iterator_t p);
12. iterator_t begin();
13. iterator_t end();
14. }

```

- retrieve: recupera el elemento de la posición p, devolviendo una referencia al mismo, de manera que es válido hacer tanto $x = L.retrieve(p)$ como $L.retrieve(p) = x$. Se puede aplicar a cualquier posición p dereferenciable. Si se usa en el miembro izquierdo de la igualdad, modifica a la lista ya que, al retornar una referencia al elemento, éste puede ser usado como un “valor assignable” (“left value”).
- insert: $q = L.insert(p, x)$ inserta al elemento x en la posición p de la lista L, devolviendo una posición q al elemento insertado. Todas las posiciones de p en adelante (incluyendo p) pasan a ser inválidas, por eso la función devuelve a q, la nueva posición insertada. Es válido insertar en cualquier posición dereferenciable y también en la posición ficticia $end()$.
- erase: $q = L.erase(p)$ elimina el elemento en la posición p, devolviendo una posición q al elemento que previamente estaba en la posición siguiente a p. Todas las posiciones de p en adelante pasan a ser inválidas y sólo se puede suprimir en las posiciones dereferenciables de la lista.

- next: $q = L.next(p)$ devuelve la posición del siguiente elemento, dada una posición dereferenciable p. Si p es la última posición dereferenciable, entonces devuelve la posición ficticia. No modifica la lista.
- begin: $p = L.begin()$ devuelve la posición del primer elemento de la lista.
- end: $p = L.end()$ devuelve la posición ficticia (no dereferenciable), después del final de la lista.
- copiar: $q = p$ copia una posición a otra. Para comparar posiciones, no se puede por operadores de comparación, sólo por igualdad o desigualdad: $q == p$, $r != L.end()$.

POSICIONES INVALIDAS: Posiciones que se convierten en inválidas tras el uso de las funciones insert o erase, normalmente desde el punto de inserción/supresión en adelante, incluyendo $end()$.

```

1. int *min(int *v, int n) {
2. int x = v[0];
3. int jmin = 0;
4. for (int k=1; k<n; k++) {
5. if (v[k]<x) {
6. jmin = k;
7. x = v[jmin];
8. }
9. }
10. return &v[jmin];
11. }
12.
13. void print(int *v, int n) {
14. cout << "Vector: (";
15. for (int j=0; j<n; j++) cout << v[j] << " ";
16. cout << "), valor minimo: " << *min(v, n) << endl;
17. }
18.
19. int main() {
20. int v[] = {6,5,1,4,2,3};
21. int n = 6;
22.
23. print(v, n);
24. for (int j=0; j<6; j++) {
25. *min(v, n) = 2 * (*min(v, n));
26. print(v, n);
27. }
28. }

```

FUNCIONES QUE RETORNA

REFERENCIAS: Esto es, funciones que dan acceso a ciertos componentes internos de estructura complejas, permitiendo cambiar su valor.

Si se desea que una función retorne directamente un objeto modificable, es posible hacer que retorne un puntero (figura A).

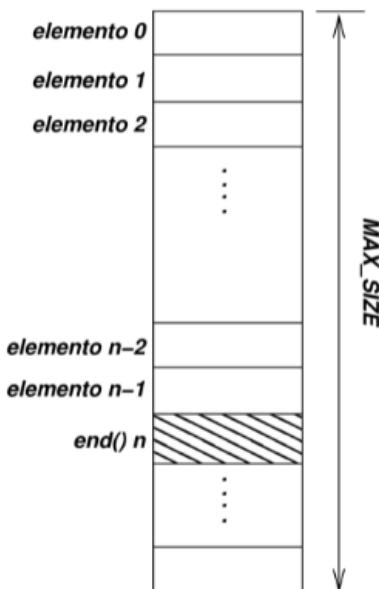
Se pueden retornar directamente referencias a los elementos (figura B), de manera que no hace falta después dereferenciarlos como en la línea 25 de la forma de puntero.

```

1. int &min(int *v, int n) {
2. int x = v[0];
3. int jmin = 0;
4. for (int k=1; k<n; k++) {
5. if (v[k]<x) {
6. jmin = k;
7. x = v[jmin];
8. }
9. }
10. return v[jmin];
11. }
12.
13. void print(int *v, int n) {
14. cout << "Vector: (";
15. for (int j=0; j<n; j++) cout << v[j] << " ";
16. cout << "), valor minimo: " << min(v, n) << endl;
17. }
18.
19. int main() {
20. int v[] = {6,5,1,4,2,3};
21. int n = 6;
22.
23. print(v, n);
24. for (int j=0; j<6; j++) {
25. min(v, n) = 2*min(v, n);
26. print(v, n);
27. }
28. }

```

IMPLEMENTACIÓN DE LISTAS POR ARREGLOS

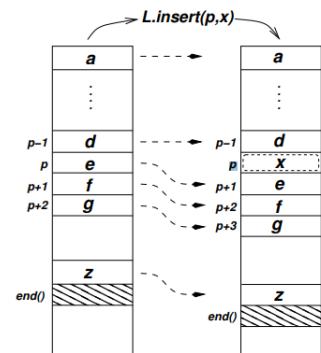


En esta representación, los valores son almacenados en celdas contiguas de un arreglo. Las posiciones se representan simplemente mediante enteros (recordemos que, en general, no es así para otras implementaciones).

DESVENTAJA: Para insertar un elemento en una posición intermedia de la lista requiere mover todos los elementos que le suceden una posición hacia el final. Igualmente, para borrar un elemento hay que desplazar todos los elementos que suceden una posición hacia el comienzo para llenar el hueco dejado por el elemento eliminado.

El almacenamiento es rígido. Si en algún momento se insertan más de MAX_SIZE elementos, se produce un error y el programa se detiene. Se puede arreglar con un arreglo `elems` y liberando el anterior, pero esto puede tener un impacto en el tiempo de ejecución si la reallocación se hace muy seguido. Además, $insert(p, x)$ y $erase(p)$ son de orden $O(n)$, pero como son de las rutinas más usadas sobre estas listas, se desea disminuir sus tiempos de ejecución, idealmente a $O(1)$.

VENTAJAS: Para el `clear()`, asigna a `size 0`, siendo así de orden $O(1)$. La alternativa genérica (`erase(begin(), end())`) sería $O(n)$. Las demás rutinas, excepto `erase` y `print`, son $O(1)$.



```

8. int list::MAX_SIZE=100;
9.
10. list::list() : elems(new elem_t[MAX_SIZE]), 
11.      size(0) {} 
12.
13. list::~list() { delete[] elems; }
14.
15. elem_t &list::retrieve(iterator_t p) {
16.     if (p<0 || p>size) {
17.         cout << "p: mala posicion.\n";
18.         abort();
19.    }
20.    return elems[p];
21. }
22.
23.
24. iterator_t list::begin() { return 0; }
25.
26. iterator_t list::end() { return size; }
27.
28. iterator_t list::next(iterator_t p) {
29.     if (p<0 || p>size) {
30.         cout << "p: mala posicion.\n";
31.         abort();
32.    }
33.    return p+1;
34. }
35.
36. iterator_t list::prev(iterator_t p) {
37.     if (p<0 || p>size) {
38.         cout << "p: mala posicion.\n";
39.         abort();
40.    }
41.    return p-1;
42. }
43.
44. iterator_t list::insert(iterator_t p, elem_t k) {
45.     if (size>=MAX_SIZE) {
46.         cout << "La lista esta llena.\n";
47.         abort();
48.    }
49.    if (p<0 || p>size) {
50.        cout << "Insertando en posicion invalida.\n";
51.        abort();
52.    }
53.    for (int j=size; j>p; j--) elems[j] = elems[j-1];
54.    elems[p] = k;

```

TIEMPOS DE EJECUCIÓN

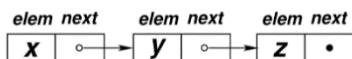
- insert(p,x), erase(p): O(n) [T = c(n-j)], donde j es la posición entera de la posición abstracta p.
- erase(p,q): O(n) [T = c(n-k)]
- clear(): O(1)
- begin(), end(), next(), retrieve(): O(1).
- prev(p): O(1)

IMPLEMENTACIÓN DE LISTAS MEDIANTE CELDAS ENLAZADAS POR PUNTEROS

```

1. class cell {
2.     friend class list;
3.     elem_t elem;
4.     cell *next;
5.     cell() : next(NULL) {}
6. };

```



Esta implementación es la más conocida y la más usada. Teniendo un puntero a una de las celdas, es fácil seguir la cadena de enlaces en la dirección de los links y recorrer todas las celdas siguientes; sin embargo, es imposible recorrer la celda en el sentido contrario. Si tenemos un puntero a la celda que contiene a z, no podemos saber cual es la celda cuyo campo next apunta a ella, es decir, la que contiene y.

Las celdas normalmente son alojadas con new y liberadas con delete, es decir que están en el área de almacenamiento dinámico del programa. Luego, hay que tener cuidado en liberar la memoria alocada para no producir pérdidas de memoria.

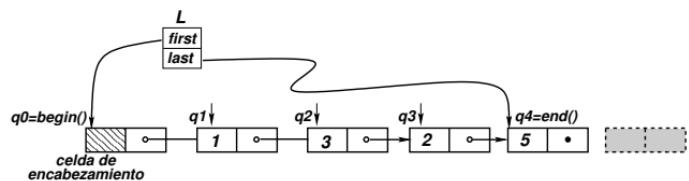
La lista se termina cuando se encuentra el “terminador”, es decir, un NULL en el campo next. Está garantizado que NULL es un puntero inválido.

- **TIPO POSICIÓN (ITERATOR_T)**: Puntero a la celda anterior a la que contiene el dato, ya que si definimos como posición el puntero a la celda que contiene el dato, entonces no hay forma de obtener el puntero a la posición anterior. Luego, las posiciones están “adelantadas” con respecto a los elementos. Notar que después de la inserción, la posición de y pasa a ser el puntero a la nueva celda s:

- **CELDA DE ENCABEZAMIENTO**: La posición del primer elemento se resuelve introduciendo una celda que no contiene dato y tal que first apunta a ella, luego su puntero es el tipo posición del primer elemento reconocible. La lista en sí es un objeto que consiste en dos punteros, uno a la primera celda (de encabezamiento) y otro a la última: la última corresponde a la posición no-dereferenciable L.end();

- **POSICIONES BEGIN() Y END()**: begin() es la posición correspondiente al primer elemento, es decir, un puntero a la celda de encabezamiento. end() es una posición ficticia después del último elemento, es decir, un puntero a la celda que contiene el último elemento.

begin() no cambia ya que ninguna inserción o supresión modifica la celda de encabezamiento; sin embargo, end() sí cambia para estos casos. End debe tener costo O(1) ya que se usa frecuentemente en los lazos para detectar el fin de la lista, entonces se mantiene un puntero a la última celda que se actualiza cuando es cambiado con insert() y erase(). Como peor caso, su complejidad algorítmica es de O(n).



```

1. list::list() : first(new cell), last(first) {
2.   first->next = NULL;
3. }
4.
5. list::~list() { clear(); delete first; }
6.
7. elem_t &list::retrieve(iterator_t p) {
8.   return p->next->elem;
9. }
10.
11. iterator_t list::next(iterator_t p) {
12.   return p->next;
13. }
14.
15. iterator_t list::prev(iterator_t p) {
16.   iterator_t q = first;
17.   while (q->next != p) q = q->next;
18.   return q;
19. }
20.
21. iterator_t
22. list::insert(iterator_t p, elem_t k) {
23.   iterator_t q = p->next;
24.   iterator_t c = new cell;
25.   p->next = c;
26.   c->next = q;
27.   c->elem = k;
28.   if (q==NULL) last = c;
29.   return p;
30. }
31.
32. iterator_t list::begin() { return first; }
33.
34. iterator_t list::end() { return last; }
35.
36. iterator_t list::erase(iterator_t p) {
37.   if (p->next==last) last = p;
38.   iterator_t q = p->next;
39.   p->next = q->next;
40.   delete q;
41.   return p;
42. }
43.
44. iterator_t list::erase(iterator_t p, iterator_t q) {
45.   if (p==q) return p;
46.   iterator_t s, r = p->next;
47.   p->next = q->next;
48.   if (!p->next) last = p;
49.   while (r!=q->next) {
50.     s = r->next;
51.     delete r;
52.     r = s;
53.   }
54.   return p;
55. }
56.
57. void list::clear() { erase(begin(),end()); }
58.
59. void list::print() {
60.   iterator_t p = begin();
61.   while (p!=end()) {
62.     cout << retrieve(p) << " ";
63.     p = next(p);
64.   }
65.   cout << endl;
66. }
67.
68. void list::printd() {
69.   cout << "h(" << first << ")";
70.   iterator_t c = first->next;
71.   int j=0;
72.   while (c!=NULL) {
73.     cout << j++ << "(" << c << ") :" << c->elem << endl;
74.     c = c->next;
75.   }
76. }
77.
78. int list::size() {
79.   int sz = 0;
80.   iterator_t p = begin();
81.   while (p!=end()) {
82.     sz++;
83.     p = next(p);
84.   }
85.   return sz;
86. }

```

erase(p,q) puede ser a favor de punteros/cursores (cuando se borran pequeños intervalos en la mitad de la lista) o a favor de los arreglos (cuando se borran grandes regiones cerca del final).

TIEMPOS DE EJECUCIÓN

- insert(p,x), erase(p): O(1).
- erase(p,q): O(n) [T = c(k-j)], donde k y j corresponden a p y q.
- clear(): O(n)
- begin(), end(), next(), retrieve(): O(1).
- prev(p): O(n) [T = cj]

IMPLEMENTACIÓN DE LISTAS MEDIANTE CELDAS ENLAZADAS POR CURSORES

```

1. class list;
2. typedef int iterator_t;
3.
4. class cell {
5.   friend class list;
6.   elem_t elem;
7.   iterator_t next;
8.   cell();
9. };
10.
11. class list {
12. private:
13.   friend class cell;
14.   static iterator_t NULL_CELL;
15.   static int CELL_SPACE_SIZE;
16.   static cell *cell_space;
17.   static iterator_t top_free_cell;
18.   iterator_t new_cell();
19.   void delete_cell(iterator_t c);
20.   iterator_t first, last;
21.   void cell_space_init();

```

Es una implementación similar a la de punteros, sólo que consiste en usar celdas enlazadas por cursores, es decir, celdas indexadas por punteros dentro de un gran arreglo de celdas. Este arreglo de celdas puede ser una variable global o un objeto estático de la clase, de manera que muchas listas pueden convivir en el mismo espacio de celdas.

• VENTAJAS CON RESPECTO A PUNTEROS:

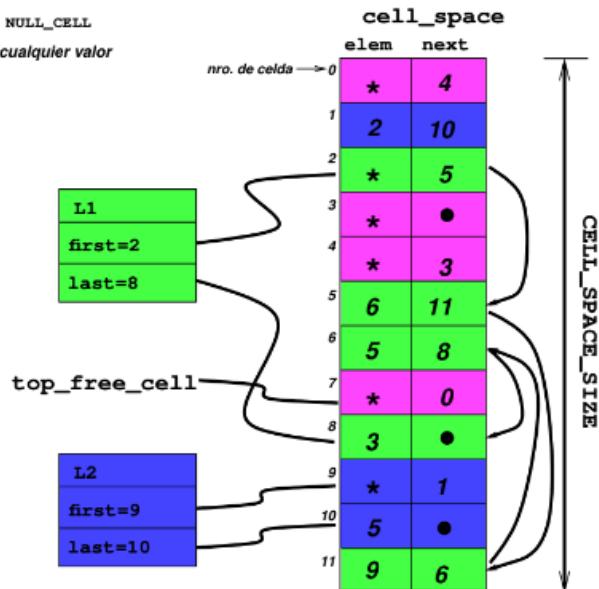
- La gestión de celdas puede llegar a ser más eficiente que la del sistema (en tiempo y memoria).
- Cuando se alojan dinámicamente con new y delete muchos objetos pequeños (como las celdas) el área de la memoria ocupada queda muy fragmentada, lo cual impide la alocación de objetos grandes, incluso después de eliminar gran parte de estos objetos pequeños. Usando cursores, todas las celdas viven en un gran espacio de celdas, de manera que no se mezclan con el resto de los objetos del programa.
- En lenguajes donde no existe alocación dinámica de la memoria, el uso de cursores reemplaza al de punteros.
- El uso de cursores ayuda a entender cómo funciona el área de almacenamiento dinámico ("heap" o "free store").

• DESVENTAJAS:

- Hay que reservar de entrada un gran espacio de celdas. Si este espacio es pequeño, corremos riesgo de que el espacio se llene y el programa aborte por la imposibilidad de alojar nuevas celdas dentro del espacio. Si es muy grande, estaremos alojando memoria que en la práctica no será usada. (Esto se puede resolver parcialmente, realocando el espacio de celdas, de manera que pueda crecer o reducirse.)
- Listas de elementos del mismo tipo comparten el mismo espacio de celdas, pero si son de diferentes tipos se debe generar un espacio de celdas por cada tipo. Esto puede agravar más aún las desventajas del punto anterior.

- Las celdas son como en el caso de los punteros, pero ahora el campo next es de tipo entero, así como las posiciones (iterator_t).
- Las celdas viven en el arreglo cell_space, que es un arreglo estándar de elementos de tipo cell. Cell_space es declarado static para que actúe como si fuera global pero dentro de la clase.
- Se declara static el tamaño del arreglo CELL_SPACE_SIZE.
- Cursor inválido: NULL_CELL = -1

● = NULL_CELL
* = cualquier valor



Las listas consistirán entonces en una serie de celdas dentro del arreglo, con una celda de encabezamiento y terminadas por una celda cuyo campo next posee el cursor inválido NULL_CELL (punto negro).

En este caso la celda de encabezamiento es el 2 que pertenece a la L1. Como el dato en las celdas de encabezamiento es irrelevante, ponemos un *.

- Gestión de celdas:

Debemos generar un sistema de gestión de celdas, es decir, escribir funciones que emulen new (new_cell) y delete (delete_cell).

Se debe mantener una lista de celdas enlazadas con las celdas libres. En el gráfico se ve que las fucsias están enlazadas. El cursor top_free_cell (estático) apunta a la primera celda libre (el 7). Luego se enlaza mediante el campo next, siendo que el campo next de la i-ésima celda libre apunta a la i+1-ésima celda libre, mientras que la última contiene un cursor inválido NULL_CELL en su next.

Es como una lista normal enlazada pero sin posiciones (es una pila). No hace falta un cursor a la última celda, sino que sólo se accede a través de uno de los extremos de la lista, apuntado por top_free_cell.

	Punteros	Cursos
Área de almacenamiento	heap	cell_space
Tipo usado para las direcciones de las celdas (iterator_t)	cell* c	int c
Dereferenciación de direcciones (dirección → celda)	*c	cell_space[c]
Dato de una celda dada su dirección c	c->elem	cell_space[c].elem
Enlace de una celda (campo next) dada su dirección c	c->next	cell_space[c].next
Alocar una celda	c = new_cell();	c = new_cell();
Liberar una celda	delete_cell();	delete_cell(c);
Dirección inválida	NULL	NULL_CELL

Las funciones c = new_cell() y delete_cell() definen una interfaz abstracta dentro de la clase list para la gestión de celdas. La primera devuelve una nueva celda libre y la segunda libera una celda utilizada, en forma equivalente a new y delete.

Antes de hacer cualquier operación sobre una lista, debemos asegurarnos que el espacio de celdas esté correctamente inicializado. Esto se hace dentro de la función cell_space_init(), la cual aloca el espacio celdas e inserta todas las celdas en la lista de celdas libres. Esto se hace en un lazo (líneas 16-17) en el cual se enlaza la celda i con la i+1, mientras que en la última celda CELL_SPACE_SIZE-1 se inserta el cursor inválido.

Para que cell_space_init() sea llamado automáticamente antes de cualquier operación, se incluye en el constructor de la clase lista (línea 9). Podemos verificar si el espacio ya fue inicializado por el valor del puntero cell_space ya que si el espacio no fue inicializado entonces este puntero es nulo (ver línea 3).

```

1. cell::cell() : next(list::NULL_CELL) {}
2.
3. cell *list::cell_space = NULL;
4. int list::CELL_SPACE_SIZE = 100;
5. iterator_t list::NULL_CELL = -1;
6. iterator_t list::top_free_cell = list::NULL_CELL;
7.
8. list::list() {
9.     if (!cell_space) cell_space_init();
10.    first = last = new_cell();
11.    cell_space[first].next = NULL_CELL;
12. }
13.
14. void list::cell_space_init() {
15.     cell_space = new cell[CELL_SPACE_SIZE];
16.     for (int j=0; j<CELL_SPACE_SIZE-1; j++)
17.         cell_space[j].next = j+1;
18.     cell_space[CELL_SPACE_SIZE-1].next = NULL_CELL;
19.     top_free_cell = 0;
20. }
21.
22. iterator_t list::new_cell() {
23.     iterator_t top = top_free_cell;
24.     if (top==NULL_CELL) {
25.         cout << "No hay mas celdas \n";
26.         abort();
27.     }
28.     top_free_cell = cell_space[top_free_cell].next;
29.     return top;
30. }
31.
32. void list::delete_cell(iterator_t c) {
33.     cell_space[c].next = top_free_cell;
34.     top_free_cell = c;
35. }
36.
37. list::~list() { clear(); }
38.
39. elem_t &list::retrieve(iterator_t p) {
40.     iterator_t q= cell_space[p].next;
41.     return cell_space[q].elem;
42. }
43.
44. iterator_t list::next(iterator_t p) {
45.     return cell_space[p].next;
46. }
47.
48. iterator_t list::prev(iterator_t p) {
49.     iterator_t q = first;
50.     while (cell_space[q].next != p)
51.         q = cell_space[q].next;
52.     return q;
53. }
54.
55. iterator_t list::insert(iterator_t p, elem_t k) {
56.     iterator_t q = cell_space[p].next;
57.     iterator_t c = new_cell();
58.     cell_space[p].next = c;
59.     cell_space[c].next = q;
60.     cell_space[c].elem = k;
61.     if (q==NULL_CELL) last = c;
62.     return p;
63. }
64.
65. iterator_t list::begin() { return first; }
66.
67. iterator_t list::end() { return last; }
68.
69. iterator_t list::erase(iterator_t p) {
70.     if (cell_space[p].next == last) last = p;
71.     iterator_t q = cell_space[p].next;
72.     cell_space[p].next = cell_space[q].next;
73.     delete_cell(q);
74.     return p;
75. }
76.
77. iterator_t list::erase(iterator_t p, iterator_t q) {
78.     if (p==q) return p;
79.     iterator_t s, r = cell_space[p].next;
80.     cell_space[p].next = cell_space[q].next;
81.     if (cell_space[p].next == NULL_CELL) last = p;
82.     while (r!=cell_space[q].next) {
83.         s = cell_space[r].next;

```

TIEMPOS DE EJECUCIÓN

- insert(p,x), erase(p): O(1).
- erase(p,q): O(n) [T = c(k-j)], donde k y j corresponden a p y q.
- clear(): O(n)
- begin(), end(), next(), retrieve(): O(1).
- prev(p): O(n) [T = cj]

INTERFAZ STL

• TEMPLATES:

En la forma en que hemos visto, la lista sólo puede tener un tipo de elemento dado, listas de enteros o de dobles, pero no de las dos a un mismo tiempo. El tipo se puede definir insertando una línea `typedef elem_t int` al principio.

Si queremos manipular listas de diferente tipo, entonces se debe duplicar el código y definir tipos `list_int` y `list_double`, lo mismo con los iteradores `list_int_iterator` y `list_double_iterator`. Esto no es deseable ya que lleva a una duplicación de código completamente innecesaria. Luego, se utilizan “**templates**” (definir clases o funciones parametrizadas por un tipo (u otros objetos también)): `list<int> lista_1; list<double> lista_2;`

Si usamos templates, deberemos tener un template separado para la clase de iteradores, por ejemplo, `iterator<int>`. Pero cuando tengamos otros contenedores como conjuntos (set) y correspondencias (map), cada uno tendrá su clase de posiciones y para evitar la colisión, podemos agregarle el tipo de contenedor al nombre de la clase, por ejemplo, `list_iterator<int>`, `set_iterator<int>` o `map_iterator<int>`.

Esto puede evitarse haciendo que la clase iterator sea una **clase anidada** de su contenedor, es decir que la declaración de la clase iterator está dentro de la clase del contenedor correspondiente y el contenedor actúa como un namespace para la clase del contenedor: `list<int>::iterator`, `set<int>::iterator`.

El anidamiento (“nesting”) de clases no tiene nada que ver con la amistad (“friendship”) entre ellas. De manera que debemos declarar en cada una de ellas como friend a la otra.

• OPERADORES DE INCREMENTO PREFIJO Y POSTFIJO:

`q = p++;` es equivalente a `q = p;` `p = p.next();` `q = ++p;` es equivalente a `p = p.next();` `q = p.`

Por ejemplo, si tengo suma `+= *p++`, estoy incrementando la variable suma con el elemento de la posición `p` antes de incrementar `p`. Si fuera suma `+= *++p`, incrementaría `p` y luego incrementaría la variable suma con el elemento de la posición `p`.

```
1. #ifndef AED_LIST_H
2. #define AED_LIST_H
3.
4. #include <cstddef>
5. #include <iostream>
6.
7. namespace aed {
8.
9.     template<class T>
10.    class list {
11.        public:
12.            class iterator;
13.        private:
14.            class cell {
15.                friend class list;
16.                friend class iterator;
17.                T t;
18.                cell *next;
19.                cell() : next(NULL) {}
20.            };
21.            cell *first, *last;
22.        public:
23.            class iterator {
24.                private:
25.                    friend class list;
26.                    cell *ptr;
27.                public:
28.                    T & operator*() { return ptr->next->t; }
29.                    T *operator->() { return &ptr->next->t; }
30.                    bool operator!=(iterator q) { return ptr!=q.ptr; }
31.                    bool operator==(iterator q) { return ptr==q.ptr; }
32.                    iterator(cell *p=NULL) : ptr(p) {}
33. // Prefix:
34.                    iterator operator++() {
35.                        ptr = ptr->next;
36.                        return *this;
37.                    }
38. // Postfix:
39.                    iterator operator++(int) {
40.                        iterator q = *this;
41.                        ptr = ptr->next;
42.                        return q;
43.                    }
44.    };
45.
46.    list() {
47.        first = new cell;
48.        last = first;
49.    }
50.    ~list() { clear(); delete first; }
51.    iterator insert(iterator p,T t) {
52.        cell *q = p.ptr->next;
53.        cell *c = new cell;
54.        p.ptr->next = c;
55.        c->next = q;
56.        c->t = t;
57.        if (q==NULL) last = c;
58.        return p;
59.    }
60.    iterator erase(iterator p) {
61.        cell *q = p.ptr->next;
62.        if (q==last) last = p.ptr;
63.        p.ptr->next = q->next;
64.        delete q;
65.        return p;
66.    }
67.    iterator erase(iterator p,iterator q) {
68.        cell *s, *r = p.ptr->next;
69.        p.ptr->next = q.ptr->next;
70.        if (!p.ptr->next) last = p.ptr;
71.        while (r!=q.ptr->next) {
72.            s = r->next;
73.            delete r;
74.            r = s;
75.        }
76.        return p;
77.    }
78.    void clear() { erase(begin(),end()); }
79.    iterator begin() { return iterator(first); }
80.    iterator end() { return iterator(last); }
81.    void print() {
82.        iterator p = begin();
83.        while (p!=end()) std::cout << *p++ << " ";
84.        std::cout << std::endl;
85.    }
86.    void printd() {
87.        std::cout << "h(" << first << ")" << std::endl;
88.        cell *c = first->next;
89.        int j=0;
90.        while (c!=NULL) {
91.            std::cout << j++ << "(" << c << ")" : " << c->t << std::endl;
92.            c = c->next;
93.        }
94.    }
95.    int size() {
96.        int sz = 0;
97.        iterator p = begin();
98.        while (p++!=end()) sz++;
99.        return sz;
100.    }
101.};
102.
103.}
104. #endif
```

Sobrecarga de operadores

```
p = L.next(p); → p++
p = L.prev(p); → p--
x = L.retrieve(p); → x = *p
```

```
1. class cell {
2.     elem_t elem;
3.     cell *next, *prev;
4.     cell() : next(NULL), prev(NULL) {}
5. }
```

LISTAS DOBLEMENTE ENLAZADAS

Si es necesario realizar repetidamente la operación `q = L.prev(p)` que retorna la posición `q` anterior a `p` en la lista `L`, entonces probablemente convenga utilizar una “lista doblemente enlazada”, como lo es `list<>` de STL.

En este tipo de listas cada celda tiene dos punteros, uno al elemento siguiente y otro al anterior.

VENTAJA: La posición puede implementarse como un “punto a la celda que contiene al elemento” y no a la celda precedente. Además, las operaciones sobre la lista pasan a ser completamente simétricas en cuanto al principio y al fin de la lista.

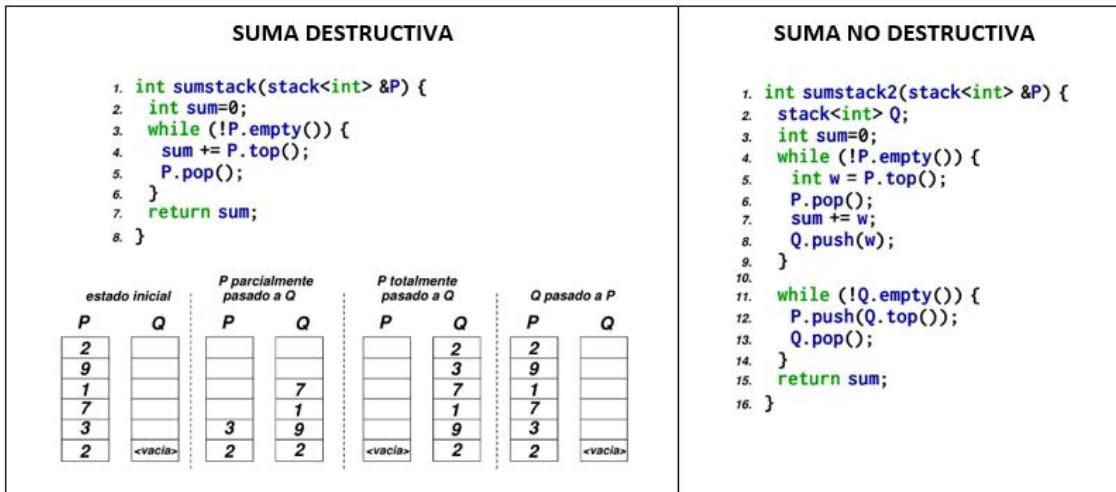
TEMPLATE LIST<CLASS T>

- Hemos incluido un namespace `aed`. Por lo tanto, las clases deben ser referenciadas como `aed::list<int>` y `aed::list<int>::iterator`. O usar: `using namespace aed;`
- `template<class T>` indica que la clase `T` es un tipo genérico. Al hacer `list<int>`, el tipo genérico `T` pasa a tomar el valor concreto `int`.
- `iterator<T>` y `cell<T>` son clases anidadas dentro de la clase `list<T>`. Evita tener que agregarle un prefijo o sufijo (como en `list_cell`, `stack_cell`).
- `cell<T>` declara friend a `list<T>` e `iterator<T>` (anidamiento no implica amistad). Si queremos que `list<T>` acceda a los miembros privados de `cell<T>` e `iterator<T>` entonces debemos declarar a las list como friend en `cell` e `iterator`. En este caso sólo es necesario que `cell` declare friend a `iterator` y `list` y que `iterator` declare a `list`.
- Las clases `list<T>`, `cell<T>` e `iterator<T>` son un ejemplo de “clases fuertemente ligadas” (“tightly coupled classes”): una serie de grupo de clases que están conceptualmente asociadas y probablemente son escritas por el mismo programador, luego se levantan todas las restricciones con declaraciones friend.
- La clase `cell<T>` es declarada privada dentro de `list<T>` ya que normalmente no debe ser accedida por el usuario de la clase, quien accede mediante `iterator<T>`.
- Para sobrecargar los operadores de incremento y dereferenciación debemos declarar a `iterator<T>` como una clase y no como un `typedef` en la interfaz básica. `iterator` contiene como único miembro un puntero a celda un `cell *ptr` de manera que: `iterator_t q = p -> next` se convierte en `cell *q = p.ptr -> next`.

TIPOS DE DATOS ABSTRACTOS FUNDAMENTALES: TAD PILA

El **TAD pila** es un subtipo de la lista, donde todas las operaciones se realizan en un extremo de la lista: el tope. Se le llama estructura “LIFO” (“Last In, First Out”), es decir, “el último en entrar es el primero en salir” como en un stack de libros.

- Sus **operaciones abstractas** son: insertar un elemento en el tope de la pila, obtener el valor del elemento en el tope de la pila, eliminar el elemento del tope.
- INTERFAZ:** Es compatible con STL y casi todas las librerías que implementan pilas usan una interfaz similar a esta: `elem_t top(); void pop(); void push(elem_t x);`. Se han agregado tres funciones auxiliares: `void clear(); int size(); bool empty();`



- Implementación de pilas mediante listas:** La pila se puede implementar fácilmente a partir de una lista, asumiendo que el tope de la pila está en el comienzo de la lista. `push(x)` y `pop()` se pueden implementar a partir de `insert` y `erase` en el comienzo de la lista, mientras que `top()` se puede implementar en base a `retrieve`.

La implementación por punteros y cursores es apropiada, ya que todas estas operaciones son $O(1)$. Notar que si se usa el último elemento de la lista como tope, entonces las operaciones de `pop()` pasan a ser $O(n)$ ya que, asumiendo que contamos con la posición q del último elemento de la lista, al hacer un `pop()`, q deja de ser válida y para obtener la nueva posición del último elemento de la lista debemos recorrerla desde el principio. Por otra parte, en la implementación de listas por arreglos tanto la inserción como la supresión en el primer elemento son $O(n)$. En cambio, si podría implementarse con una implementación basada en arreglos si el tope de la pila está al final, ya que la inserción y la supresión en el último elemento son $O(1)$: El tamaño de la pila es guardado en un miembro `int size_m`. Este contador es inicializado a cero en el constructor y después es actualizado durante las operaciones que modifican la longitud de la lista como `push()`, `pop()` y `clear()`.

A continuación, declaraciones e implementaciones de una pila basada en listas respectivamente:

```

1. class stack : private list {
2. private:
3.     int size_m;
4. public:
5.     stack();
6.     void clear();
7.     elem_t& top();
8.     void pop();
9.     void push(elem_t x);
10.    int size();
11.    bool empty();
12. };

```

```

1. stack::stack() : size_m(0) { }
2. elem_t& stack::top() {
3.     return retrieve(begin());
4. }
5. void stack::pop() {
6.     erase(begin()); size_m--;
7. }
8. void stack::push(elem_t x) {
9.     insert(begin(),x); size_m++;
10. }
```

Método	Lista(begin) [1]	Lista(end) [2]	Arreglo [3]
push(x)	$O(1)$	$O(1)$	$O(1)$
pop(), top()	$O(1)$	$O(n)$	$O(1)$
clear()	$O(n)$	$O(n)$	$O(1)$

- 1- Lista por punteros o cursores, con `top()` en `begin()`
- 2- Lista por punteros o cursores, con `top()` en `end()`
- 3- Arreglos con `top()` al final del arreglo.

```

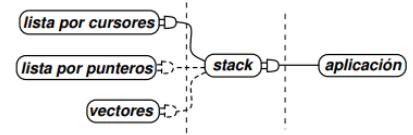
1. #ifndef AED_STACK_H
2. #define AED_STACK_H
3.
4. #include <aedsrc/list.h>
5.
6. namespace aed {
7.
8.     template<class T>
9.     class stack : private list<T> {
10.     private:
11.         int size_m;
12.     public:
13.         stack() : size_m(0) { }
14.         void clear() { erase(begin(),end()); size_m = 0; }
15.         T &top() { return *begin(); }
16.         void pop() { erase(begin()); size_m--; }
17.         void push(T x) { insert(begin(),x); size_m++; }
18.         int size() { return size_m; }
19.         bool empty() { return size_m==0; }
20.     };
21. 
```

Clase lista con templates (implementación STL). No hay mucha diferencia porque no tiene iteradores.

- La pila como adaptador:** La pila deriva directamente de la lista, pero con una declaración `private`. De esta forma, el usuario de la clase `stack` no puede usar métodos de la clase `lista`.

El hecho de que la pila sea tan simple permite que pueda ser implementada en términos de otros contenedores también, como por ejemplo el vector de STL. De ahí el término “adaptador”, pues brinda un subconjunto de la funcionalidad del contenedor original.

VENTAJA: La ventaja de operar sobre el adaptador (la pila) y no directamente sobre el contenedor básico (la lista) es que el adaptador puede después conectarse fácilmente a otros contenedores (representado por enchufes en la figura).



TIPOS DE DATOS ABSTRACTOS FUNDAMENTALES: TAD COLA

Por contraposición con la pila, la **cola** es un contenedor de tipo “FIFO” (First In, First Out: el primero en entrar es el primero en salir). La cola es un objeto muchas veces usado como buffer o pulmón, es decir, un contenedor donde almacenar una serie de objetos que deben ser procesados, manteniendo el orden en el que ingresaron. La cola es, como la pila, un subtipo de lista y llama también a ser implementada como un adaptador.

- Sus **operaciones abstractas** son: obtener el elemento en el frente de la cola, eliminar el elemento en el frente de la cola, agregar un elemento a la cola.

- INTERFAZ:** La cola no tiene posiciones, de manera que no necesita clases anidadas ni sobrecarga de operadores, por lo que el código es sencillo y presentamos directamente la versión compatible STL.

Las operaciones abstractas se realizan a través de `pop()` y `front()`, que operan sobre el principio de la lista, y `push()` que opera sobre el fin de la lista. Todas estas operaciones son $O(1)$.

También se agregan operaciones estándar `size()` y `empty()` que son $O(1)$ y `clear()`, $O(n)$.

```
1. template<class T>
2. class queue : private list<T> {
3. private:
4.     int size_m;
5. public:
6.     queue() : size_m(0) { }
7.     void clear() { erase(begin(), end()); size_m = 0; }
8.     T &front() { return *begin(); }
9.     void pop() { erase(begin()); size_m--; }
10.    void push(T x) { insert(end(), x); size_m++; }
11.    int size() { return size_m; }
12.    bool empty() { return size_m == 0; }
13. }
```

TIPOS DE DATOS ABSTRACTOS FUNDAMENTALES: TAD COLA

La **correspondencia** o *memoria asociativa* es un contenedor que almacena la relación entre elementos de un cierto conjunto universal D llamado “**dominio**” con elementos de otro conjunto universal llamado el “**contradominio**” o “rango”. Se utiliza la terminología “**clave**” (“key”) para un valor del dominio y “**valor**” para referirse a los elementos del contradominio.

Un dato elemento del dominio o bien no debe tener asignado ningún elemento del contradominio o bien debe tener asignado uno solo. Por otra parte, puede ocurrir que a varios elementos del dominio se les asigne un solo elemento del contradominio.

- USO Y DESUSO:** En general, se guardan internamente pares clave/valor y poseen algún algoritmo para asignar valores a claves, en forma análoga a cómo funcionan las bases de datos para luego recuperar rápidamente la asignación. Luego, para $j \rightarrow j^2$ es mucho más eficiente usar una función, ya que es mucho más rápido y no es necesario almacenar todos los valores.

El uso de un contenedor tipo correspondencia es útil cuando no es posible calcular el elemento del contradominio a partir del elemento del dominio. Por ejemplo, un tal caso es una correspondencia entre el número de documento de una persona y su nombre.

- OPERACIONES ABSTRACTAS:** Consultar la correspondencia para saber si una dada clave tiene un valor asignado, asignar un valor a una clave y recuperar el valor asignado a una clave.

- INTERFAZ SIMPLE:**

```
1. class iterator_t /* ... */;
2.
3. class map {
4. private:
5.     // ...
6. public:
7.     iterator_t find(domain_t key);
8.     iterator_t insert(domain_t key, range_t val);
9.     range_t& retrieve(domain_t key);
10.    void erase(iterator_t p);
11.    int erase(domain_t key);
12.    domain_t key(iterator_t p);
13.    range_t& value(iterator_t p);
14.    iterator_t begin();
15.    iterator_t next(iterator_t p);
16.    iterator_t end();
17.    void clear();
18.    void print();
19. }
```

Está basada en la interfaz STL, pero por simplicidad evitamos el uso de clases anidadas para el correspondiente iterator y también evitamos el uso de templates y sobrecarga de operadores. Primero se deben definir (probablemente via `typedef's`) los tipos que corresponden al dominio (`domain_t`) y al contradominio (`range_t`).

Una clase iterator representa las posiciones en la correspondencia. Sin embargo, considerar de que, en contraposición con las listas y a semejanza de los conjuntos, no hay un orden definido entre los pares de la correspondencia. Por otra parte, en la correspondencia el iterator itera sobre *los pares de valores* que representan la correspondencia.

M es una correspondencia, p es un iterator, k una clave (tipo `domain_t`), val un elemento del contradominio (tipo `range_t`). Luego, las operaciones de la clase son:

- **p = M.find(k):** Dada una clave k, devuelve un iterator al par correspondiente (si existe debe ser único). Si k no tiene asignado ningún valor, devuelve `end()`.
- **p = M.insert(k, val):** Asigna a k el valor val. Si k ya tenía asignado un valor, lo reemplaza; si no tenía, lo asigna. Retorna un iterator al par.
- **val = M.retrieve(k):** Recupera el valor asignado a k. Si k no tiene ningún valor, inserta una asignación de k al valor creado por defecto para el tipo `range_t` (!!!).
- **m.retrieve(k) = val;** Esta función se puede usar como miembro izquierdo ya que retorna una referencia al valor asignado a k.
- **k = M.key(p):** Retorna la clave correspondiente a la asignación apuntada por p. No retorna una referencia, no se puede usar como miembro izquierdo.
- **val = M.value(p):** Retorna el valor correspondiente a la asignación apuntada por p. Es una referencia, de manera que podemos usarlo como miembro izquierdo.
- **erase(p):** Elimina la asignación apuntada por p (debe ser dereferenciable, esto es, verificar que p sea distinta del `end()`).
- **n = erase(k):** Elimina la asignación correspondiente a k (si la hay). Retorna el número de asignaciones efectivamente eliminadas (0 o 1).
- **p = M.begin():** Retorna un iterator a la primera asignación (en un orden no especificado).
- **p = M.end():** Retorna un iterator a una asignación ficticia después de la última (en un orden no especificado).
- **clear():** Elimina todas las asignaciones.

- **IMPLEMENTACIÓN MEDIANTE CONTENEDORES LINEALES:** Se guarda en un contenedor todas las asignaciones. A las listas y vectores se les llama contenedores lineales, ya que en ellos existe un ordenamiento natural de las posiciones.

Definimos una clase elem_t que simplemente contiene dos campos first y second con la clave y el valor de la asignación (los nombres de los campos vienen de la clase pair de las STL). Estos pares de elementos (asignaciones) se podrían guardar tanto en un vector<elem_t> como en una lista (list<elem_t>), asumiendo las listas por punteros o cursorios.

ELEMENTOS DESORDENADOS:

- **p = find(k)** debe recorrer todas las asignaciones y si encuentra una cuya campo first coincide con k, entonces debe devolver el iterator correspondiente. Si la clave no tiene ninguna asignación, entonces después de recorrer todas las asignaciones, debe devolver end(). El peor caso de find() es cuando la clave no está asignada, o la asignación está al final del contenedor, en cuyo caso es O(n) ya que debe recorrer todo el contenedor (n es el número de asignaciones en la correspondencia). Si la clave tiene una asignación, entonces el costo es proporcional a la distancia desde la asignación hasta el origen: el mejor caso es O(1) y el costo en promedio será O(n/2).
- **insert()** debe llamar inicialmente a find(). Si la clave ya tiene un valor asignado, la inserción es O(1) tanto para listas como vectores. En el caso de vectores, notar que esto se debe a que no es necesario insertar una nueva asignación. Si la clave no está asignada, entonces el elemento se puede insertar al final para vectores, lo cual también es O(1), y en cualquier lugar para listas. **retrieve(k)** es equivalente a find().
- Para **erase(p)** sí hay diferencias, la implementación por listas es O(1) mientras que la implementación por vectores es O(n) ya que implica mover todos los elementos que están después de la posición eliminada.

Operación	lista	vector
find(key)	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
insert(key, val)	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
retrieve(key)	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
erase(p)	$O(1)$	$O(1)/O(n)/O(n)$
key, value, begin, end,	$O(1)$	$O(1)$
clear	$O(n)$	$O(1)$

La notación es mejor/promedio/peor. Si los tres son iguales, se reporta uno solo.

ELEMENTOS ORDENADOS (LISTAS):

- Se reduce el tiempo de ejecución al ordenar los pares de asignación de menor a mayor, según la clave.
- Ahora p = find(k) no debe recorrer toda la lista cuando la clave no está, ya que el algoritmo puede detenerse cuando encuentra una clave mayor a la que se busca. Sin embargo, al insertar nuevas asignaciones hay que tener en cuenta que no se debe insertar en cualquier posición, sino que hay que mantener la lista ordenada.
 - Tanto p = find(key) como p = insert(key, val) se basan en una función auxiliar (en el private por ser auxiliar para insert y find) p = lower_bound(key) que retorna la primera posición donde podría insertarse la nueva clave sin violar la condición de ordenamiento sobre el contenedor. Como casos especiales, si todas las claves son mayores que key entonces debe retornar begin() y si son todas menores, o la correspondencia está vacía, debe retornar end().

```

1. class map;
2.
3. class elem_t {
4. private:
5.     friend class map;
6.     domain_t first;
7.     range_t second;
8. };
9. // iterator para map va a ser el mismo que para listas.
10. class map {
11. private:
12.     list l;
13.
14. iterator_t lower_bound(domain_t key) {
15.     iterator_t p = l.begin();
16.     while (p!=l.end()) {
17.         domain_t dom = l.retrieve(p).first;
18.         if (dom >= key) return p;
19.         p = l.next(p);
20.     }
21.     return l.end();
22. }
23.
24. public:
25.     map() {}
26.     iterator_t find(domain_t key) {
27.         iterator_t p = lower_bound(key);
28.         if (p!=l.end() && l.retrieve(p).first == key)
29.             return p;
30.         else return l.end();
31.     }
32.     iterator_t insert(domain_t key, range_t val) {
33.         iterator_t p = lower_bound(key);
34.         if (p==l.end() || l.retrieve(p).first != key) {
35.             elem_t elem;
36.             elem.first = key;
37.             p = l.insert(p,elem);
38.         }
39.         l.retrieve(p).second = val;
40.         return p;
41.     }
42.     range_t &retrieve(domain_t key) {
43.         iterator_t q = find(key);
44.         if (q==end()) q.insert(key,range_t());
45.         return l.retrieve(q).second;
46.     }
47.     bool empty() { return l.begin()==l.end(); }
48.     void erase(iterator_t p) { l.erase(p); }
49.     int erase(domain_t key) {
50.         iterator_t p = find(key); int r = 0;
51.         if (p!=end()) { l.erase(p); r = 1; }
52.         return r;
53.     }
54.     iterator_t begin() { return l.begin(); }
55.     iterator_t end() { return l.end(); }
56.     void clear() { l.erase(l.begin(),l.end()); }
57.     int size() { return l.size(); }
58.     domain_t key(iterator_t p) {
59.         return l.retrieve(p).first;
60.     }
61.     range_t &value(iterator_t p) {
62.         return l.retrieve(p).second;
63.     }
64. }
```

INTERFAZ COMPATIBLE CON STL

```

11. template<typename first_t,typename second_t>
12. class pair {
13. public:
14.     first_t first;
15.     second_t second;
16.     pair(first_t f=first_t(),second_t s=second_t())
17.         : first(f), second(s) {}
18. };
19.
20. // iterator para map va a ser el mismo que para listas.
21. template<typename domain_t,typename range_t>
22. class map {
23.
24. private:
25.     typedef pair<domain_t,range_t> pair_t;
26.     typedef list<pair_t> list_t;
27.     list_t l;
28.
29. public:
30.     typedef typename list_t::iterator iterator;
31.
32. private:
33.     iterator lower_bound(domain_t key) {
34.         iterator p = l.begin();
35.         while (p!=l.end()) {
36.             domain_t dom = p->first;
37.             if (dom >= key) return p;
38.             p++;
39.         }
40.         return l.end();
41.     }
42.
43. public:
44.     map() {}
45.
46.     iterator find(domain_t key) {
47.         iterator p = lower_bound(key);
48.         if (p!=l.end() && p->first == key)
49.             return p;
50.         else return l.end();
51.     }
52.
53. range_t &operator[](domain_t key) {
54.     iterator q = lower_bound(key);
55.     if (q==end() || q->first!=key)
56.         q = l.insert(q,pair_t(key,range_t()));
57.     return q->second;
58. }
59. bool empty() { return l.begin()==l.end(); }
60. void erase(iterator p) { l.erase(p); }
61. int erase(domain_t key) {
62.     iterator p = find(key);
63.     if (p!=end()) {
64.         l.erase(p);
65.         return 1;
66.     } else {
67.         return 0;
68.     }
69.     iterator begin() { return l.begin(); }
70.     iterator end() { return l.end(); }
71.     iterator next(iterator p) { return l.next(p); }
72.     void clear() { l.erase(l.begin(),l.end()); }
73. };
74. #endif

```

- En vez del tipo elem_t se define un template pair<class first_t, class second_t>. Este template es usado para map y otros contenedores de STL.
- Los campos first y second de pair son públicos. pair es una forma muy simple de asociar pares de valores en un único objeto, y también permite que una función retorne dos valores al mismo tiempo.
- La clase map es un template de las clases domain_t y range_t. Los elementos de la lista serán de tipo pair<domain_t, range_t>.

- Para simplificar la escritura de la clase, se definen dos tipos internos pair_t y list_t. Los elementos de la lista serán de tipo pair_t. También se define el tipo público iterator que es igual al iterator sobre la lista, pero esta definición de tipo permitirá verlo también externamente como map<domain_t, range_t>::iterator.
- Val = M.retrieve(key) se reemplaza sobrecargando el operador [], así se pueda escribir val = M[key], con el mismo efecto colateral que retrieve (asigna valor por defecto) y pudiendo usarlo como miembro izquierdo.

Operación	lista	vector
find(key)	$O(1)/O(n)/O(n)$	$O(1)/O(\log n)/O(\log n)$
M[key] (no existente)	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
M[key] (existente)	$O(1)/O(n)/O(n)$	$O(1)/O(\log n)/O(\log n)$
erase(key)	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
key, value, begin, end,	$\mathcal{O}(1)$	$\mathcal{O}(1)$
clear	$O(n)$	$O(1)$

- La notación es mejor/promedio/peor. Si los tres son iguales, $O(\dots)$.
- Si bien hay una ganancia en el caso de listas ordenadas, el orden del tiempo de ejecución es prácticamente el mismo que en el caso de listas no ordenadas.
- **find(key)** y **M[key]** están basados en **lower_bound** y tienen el mismo tiempo de ejecución ya que, para listas, las inserciones son $O(1)$.

ELEMENTOS ORDENADOS (VECTORES):

```

11. template<typename first_t,typename second_t>
12. class pair {
13. public:
14.     first_t first;
15.     second_t second;
16.     pair() : first(first_t()), second(second_t()) {}
17. };
18.
19. // iterator para map va a ser el mismo que para listas.
20. template<typename domain_t,typename range_t>
21. class map {
22.
23. public:
24.     typedef int iterator;
25.
26. private:
27.     typedef pair<domain_t,range_t> pair_t;
28.     typedef vector<pair_t> vector_t;
29.     vector_t v;
30.
31.     iterator lower_bound(domain_t key) {
32.         int p=0, q=v.size(), r;
33.         if (!q || v[p].first >key) return 0;
34.         while (q-p > 1) {
35.             r = (p+q)/2;
36.             domain_t kr = v[r].first;
37.             if (key > kr) p=r;
38.             else if (key < kr) q=r;
39.             else if (kr==key) return r;
40.         }
41.         if (v[p].first == key) return p;
42.         else return q;
43.     }
44.     public:
45.     map() {} 
46.
47.     iterator find(domain_t key) {
48.         int p = lower_bound(key);
49.         if (p == v.size() || v[p].first == key) return p;
50.         else return v.size();
51.     }
52.     range_t & operator[](domain_t key) {
53.         iterator p = lower_bound(key);
54.         if (p == v.size() || v[p].first != key) {
55.             v.push_back(pair_t());
56.             iterator q = v.size();
57.             while (~q > p) v[q] = v[q-1];
58.             v[p].first = key;
59.         }
60.         return v[p].second;
61.     }
62.     int erase(domain_t key) {
63.         iterator p = find(key); int r = 0;
64.         if (p!=end()) { erase(p); r = 1; }
65.         return r;
66.     }
67.     bool empty() { return v.size()==0; }
68.     void erase(iterator p) {
69.         iterator q = p;
70.         while (q != v.size()) {
71.             v[q] = v[q+1];
72.             q++;
73.         }
74.         v.pop_back();
75.     }
76.     iterator begin() { return 0; }
77.     iterator end() { return v.size(); }
78.     void clear() { v.clear(); }
79.     int size() { return v.size(); }
80. };
81. 
```

#endif

La ganancia real de usar contenedores ordenados es en el caso de vectores ya que en ese caso podemos usar el algoritmo “binary search” en cuyo caso $p = \text{lower_bound}(k)$ resulta ser $O(\log n)$ en el *peor caso*.

El código a la izquierda se basa en la clase vector de STL, pero en realidad con modificaciones menores se podrían reemplazar los vectores de STL por arreglos estándar de C. La correspondencia almacena las asignaciones (de tipo **pair_t**) en un **vector<pair_t> v**. Para el tipo **map<>::iterator** usamos directamente el tipo entero.

- **Lower_bound:** El algoritmo se basa en ir refinando un rango $[p,q]$ tal que la clave buscada esté garantizada siempre en el rango, es decir, $k_p \leq k < k_q$, donde k_p, k_q son las claves en las posiciones p y q respectivamente y k es la clave buscada.

Primero se verifican una serie de casos especiales hasta obtener un rango válido. Si el vector está vacío, entonces **lower_bound** retorna 0, ya que debe insertar en **end()** que en ese caso vale 0. Lo mismo si $k < k_0$, ya que en ese caso la clave va en la primera posición. Si ninguno de estos casos se aplica, entonces el rango $[p,q]$ con $p = 0$ y $m = v.size()$ es un rango válido, es decir $k_p \leq k < k_q$.

Una vez que tenemos un rango válido $[p,q]$ podemos refinarlo haciendo $r = \text{floor}((p+q)/2)$.

Esto divide $[p,q]$ en dos subrangos disjuntos $[p,r]$ y $[r,q]$, y comparando la clave k con la que está almacenada en la posición r , k_r : si $k \geq k_r$, entonces el nuevo rango es el $[r,q]$, mientras que si no es el $[p,r]$.

Tiempo de ejecución de lower_bound: Notar que si el rango $[p,q]$ es de longitud par, esto es $n = q - p$ es par, entonces los dos nuevos rangos son iguales, de longitud igual a la mitad $n = (q - p)/2$. Si el número de elementos es inicialmente una potencia de 2, digamos $n = 2^m$, entonces después del primer refinamiento la longitud será 2^{m-1} , después de dos refinamientos 2^{m-2} , hasta que, después de m refinamientos la longitud se reduce a $2^0 = 1$ y el algoritmo se detiene. Entonces $T(n) = m = \log_2 n$. Si n no es una potencia de dos, entonces el número de refinamientos es $m = \text{floor}(\log_2 n) + 1$. Mucho mejor que el $O(n)$ que teníamos con las listas.

Búsqueda binaria no se puede aplicar a listas ya que estas no son contenedores de “acceso aleatorio”. Para vectores, acceder al elemento j -ésimo es una operación $O(1)$, mientras que para listas involucra recorrer toda la lista y $O(j)$.

RELACIÓN DE ORDEN

- Para implementar la correspondencia con contenedores ordenados es necesario contar con una relación de orden en conjunto universal de las claves.
- Números básicos utilizan el orden usual, cadenas de caracteres utilizan el orden lexicográfico (alfabético).
- Para otros conjuntos universales se necesita una relación de orden:

< es una relación de orden en el conjunto C si,

- $<$ es transitiva, es decir, si $a < b$ y $b < c$, entonces $a < c$.
- Dados dos elementos cualesquiera de C, una y sólo una de las siguientes afirmaciones es válida: $a < b$, $b < a$, $a = b$.

ÁRBOLES

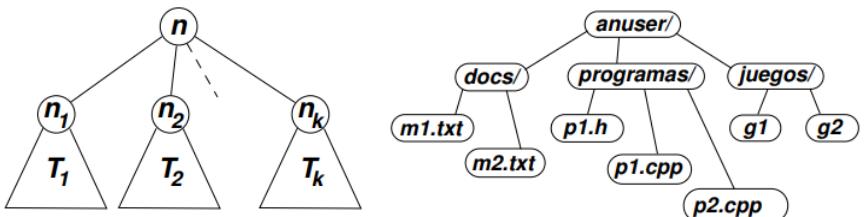
Los **árboles** son contenedores que permiten organizar un conjunto de objetos en forma jerárquica. En general sirven para representar fórmulas, la descomposición de grandes sistemas en sistemas más pequeños en forma recursiva y aparecen en forma sistemática en muchísimas aplicaciones de la computación científica.

Una de las propiedades más llamativas de los árboles es la capacidad de accedes a muchísimos objetos desde un puto de partida o raíz en unos pocos pasos. No hay un contenedor STL de tipo árbol, ya que se considera un subtipo de grafo o una entidad demasiado básica para ser utilizada directamente por los usuarios.

NOMENCLATURA BÁSICA DE ÁRBOLES: Un árbol es una colección de elementos llamados “**nodos**”, uno de los cuales es la “**raíz**”. Existe una relación de parentesco por la cual cada nodo tiene un y sólo un “**padre**”, salvo la raíz que no lo tiene. El nodo es el concepto análogo al de “**posición**” en la lista, es decir, un objeto abstracto que representa una posición en el mismo, no directamente relacionado con el “**elemento**” o “**etiqueta**” del nodo.

Un árbol se define recursivamente:

- Un nodo sólo es un árbol.
- Si n es un nodo y T_1, T_2, \dots, T_k son árboles con raíces n_1, \dots, n_k entonces podemos construir un nuevo árbol que tiene a n como raíz donde n_1, \dots, n_k son “hijos” de n.



Un “árbol vacío” se llamará Λ .

CAMINO: Si n_1, n_2, \dots, n_k es una secuencia de nodos tales que n_i es padre de n_{i+1} para $i = 1 \dots k - 1$, entonces decimos que esta secuencia de nodos es un “camino” (“path”).

La longitud de un camino es igual al número de nodos en el camino menos uno, por ejemplo: {anuser, docs, m2.txt} tiene longitud 2. Notar que siempre existe un camino de longitud 0 de un nodo a sí mismo.

DESCENDIENTES Y ANTECESORES: Si existe un camino que va del nodo a al b, entonces decimos que a es antecesor de b y b es descendiente de a.

Estrictamente hablando, un nodo es antecesor y descendiente de sí mismo ya que existe camino de longitud 0. Luego, para diferenciar, a es descendiente propio de b si a es descendiente de b, pero a \neq b.

HOJAS: Un nodo que no tiene hijos es una “hoja” del árbol. Recordemos que, por contraposición, el nodo que no tiene padres es único y es la raíz.

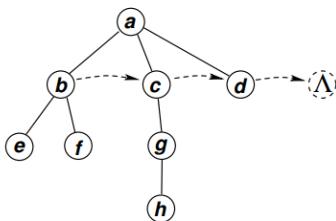
HERMANOS: Los nodos que tienen un mismo parente son hermanos entre sí. No basta con que estén en el mismo nivel, como m1.txt y p1.h.

ALTURA: Máxima longitud de un camino que va desde el nodo a una hoja. Luego, la longitud es 0 si el nodo es una hoja y $1 + \max_{\text{altura}}(\text{hijo de } n)$ si no lo es.

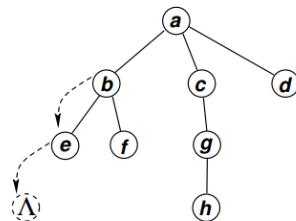
PROFUNDIDAD: Longitud del único camino que va desde el nodo a la raíz.

NIVEL: Conjunto de todos los nodos con misma profundidad.

ORDEN DE LOS NODOS: Para árboles ordenados orientados (AOO), el orden de los hijos es importante de manera que los árboles de la figura son diferentes. Si bien a tiene los mismos hijos, están en diferente orden.



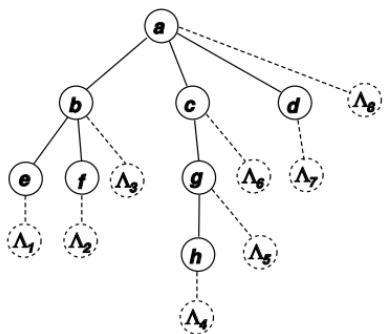
avanza por hermano derecho



avanza por hijo más izquierdo

Decimos que el nodo c está a la derecha de b, o también que c es el hermano derecho de b. También decimos que b es el hijo más a la izquierda de a. El orden entre los hermanos se propaga a los hijos, de manera que h está a la derecha de e ya que ambos son descendientes de c y b, respectivamente.

Podemos pensar al árbol como una lista bidimensional. Se puede avanzar en dos direcciones: por el hermano derecho, recorriendo la lista de hermanos de izquierda a derecha, o por el hijo más izquierdo, tratando de descender lo más posible en profundidad.



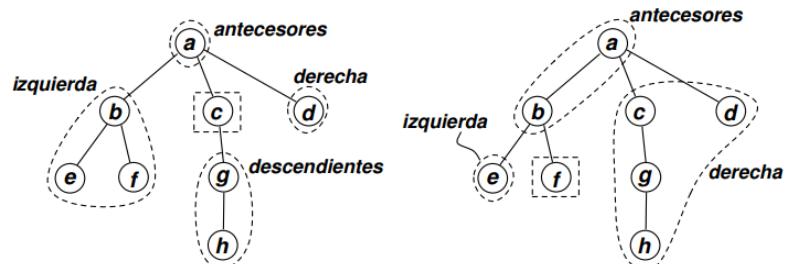
Avanzando por el hermano derecho, el recorrido termina en el último hermano a la derecha. Por analogía, con la posición `end()` en las listas, asumiremos que después del último hermano existe un nodo ficticio no dereferenciable. Además, asumiremos que el hijo más izquierdo de una hoja es un nodo ficticio no dereferenciable, simbolizado con Λ cuando dibujamos el árbol. Así, hay una posición ficticia para cada hermano derecho e hijo izquierdo que no haya en el árbol.

PARTICIONAMIENTO DEL CONJUNTO DE NODOS: Dados dos nodos cualesquiera m y n, consideremos sus caminos a la raíz. Si m es descendiente de n, entonces el camino de n está incluido en el de m o viceversa. Si entre m y n no hay relación de descendiente o antecesor, entonces los caminos se deben bifurcar necesariamente en un cierto nivel. *El orden entre m y n es el orden entre los antecesores a ese nivel*. Esto demuestra que, dados dos nodos cualesquiera m y n, sólo una de las siguientes afirmaciones puede ser cierta:

- $m = n$,
- m es antecesor propio de n o n es antecesor propio de m,
- m está a la derecha de n o n está a la derecha de m.

Luego, dado un nodo n, el conjunto N de todos los nodos del árbol se puede dividir en 5 conjuntos disjuntos:

$$N = \{n\} \cup \{\text{descendientes}(n)\} \cup \{\text{antecesores}(n)\} \cup \{\text{derecha}(n)\} \cup \{\text{izquierda}(n)\}.$$



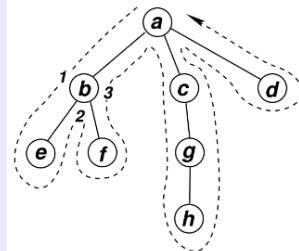
LISTADO EN ORDEN PREVIO (PREORDER): Dado un nodo n con hijos n₁, n₂, ..., n_m se define recursivamente:

$$\text{oprev}(n) = (n, \text{oprev}(n_1), \text{oprev}(n_2), \dots, \text{oprev}(n_m))$$

Además el orden previo del árbol vacío es la lista vacía: oprev(Λ) = ()

$$\begin{aligned} \text{oprev}(a) &= a, \text{oprev}(b), \text{oprev}(c), \text{oprev}(d) \\ &= a, b, \text{oprev}(e), \text{oprev}(f), c, \text{oprev}(g), d \\ &= a, b, \text{oprev}(e), \text{oprev}(f), c, \text{oprev}(g), d \\ &= a, b, e, f, c, g, \text{oprev}(h), d \\ &= a, b, e, f, c, g, h, d \end{aligned}$$

Gráficamente, se recorre el borde del árbol en el sentido contrario a las agujas del reloj. Dado un nodo como el b, el camino pasa cerca de él en varios puntos, pero el orden previo consiste en listar los nodos una sola vez, la primera vez que el camino pasa cerca del árbol.



NOTACIÓN LISP: Una expresión matemática compleja que involucra funciones cuyos argumentos son a su vez llamadas a otras funciones puede ponerse en forma de árbol. En este caso, cada función es un nodo cuyos hijos son los argumentos de la función.

Ejemplo: f(g(a,b),h(t,u,v),q(r,s(w))).

En el lenguaje Lisp la llamada a una función f(x,y,z) se escribe de la forma (f x y z), de manera que la función de arriba se escribiría como:

(f (g a b) (h t u v) (q r (s w))), de manera que el orden de los nodos es igual al del orden previo. Luego, la definición formal es:

$$\text{lisp}(n) = \begin{cases} \text{si } n \text{ es una hoja:} & n \\ \text{caso contrario:} & (n \text{ lisp}(n_1) \text{ lisp}(n_2) \dots \text{ lisp}(n_m)) \end{cases}$$

Existe una relación única entre un árbol y su notación Lisp. Los paréntesis dan la estructura adicional que permite establecer la relación única. La utilidad de esta notación es que permite fácilmente escribir árboles en una línea de texto, sin tener que recurrir a un gráfico.

También permite serializar un árbol, es decir, convertir una estructura bidimensional como es el árbol en una estructura unidimensional como es una lista. **Serializar** permite mayor simplicidad al guardar las estructuras a disco para después leerlas y enviarlas a través de la red entre diferentes procesos.

ALGORITMOS PARA LISTAR NODOS:

```

1. void preorder(tree &T, iterator n, list &L) {
2.   L.insert(L.end(), /* valor en el nodo 'n'... */);
3.   iterator c = /* hijo mas izquierdo de n... */;
4.   while (/* 'c' no es 'Lambda'... */) {
5.     preorder(T, c, L);
6.     c = /* hermano a la derecha de c... */;
7.   }
8. }

1. void postorder(tree &T, iterator n, list &L) {
2.   iterator c = /* hijo mas izquierdo de n... */;
3.   while (c != T.end()) {
4.     postorder(T, c, L);
5.     c = /* hermano a la derecha de c... */;
6.   }
7.   L.insert(L.end(), /* valor en el nodo 'n'... */);
8. }
```

LISTADO EN ORDEN POSTERIOR (POSTORDER): Dado un nodo n con hijos n₁, n₂, ..., n_m se define recursivamente:

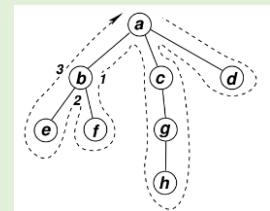
$$\text{opost}(n) = (\text{opost}(n_1), \text{opost}(n_2), \dots, \text{opost}(n_m), n)$$

De la misma manera que con el preorden, se tiene opost(Λ) = ()

$$\begin{aligned} \text{opost}(a) &= \text{opost}(b), \text{opost}(c), \text{opost}(d), a \\ &= \text{opost}(e), \text{opost}(f), b, \text{opost}(g), c, d, a \\ &= e, f, b, \text{opost}(h), g, c, d, a \\ &= e, f, b, h, g, c, d, a \end{aligned}$$

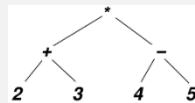
Gráficamente, se recorre el borde del árbol en sentido contrario a las agujas del reloj, listando el nodo la última vez que el recorrido pasa por al lado del mismo.

Sino se recorre en el sentido opuesto, es decir, sentido del reloj, y listando los nodos la primera vez que el camino pasa cerca de ellos. Una vez que la lista es obtenida, invertimos la lista.



NOTACIÓN POLACA INVERTIDA: Es el listado en orden posterior de árboles de expresiones. Los árboles de expresiones se forman:

- Para operadores binarios de la forma a + b se pone al operador + como padre de los dos operandos (a y b). Lo mismo para funciones binarias como rem (resto).
- Para operadores unarios (como -3) y funciones (sin(20)) se escriben poniendo el operando como hijo del operador o función.
- Para operadores asociativos con más de dos operandos deben asociarse de a 2. Esto es, 1 + 3 + 4 + 9 → ((1+3) + 4) + 9.



Luego para este árbol se tiene:
rpn = (2, 3, +, 4, 5, -, *),
el cual representa la expresión (2+3) * (4-5).

RECONSTRUCCIÓN DEL ÁRBOL: A través del orden previo y posterior puedo reconstruir el árbol (con uno sólo no puedo, sólo los dos).

oprev(n) = (n, n₁, descendientes(n₁), n₂, descendientes(n₂), ..., n_k, descendientes(n_k)),
opost(n) = (descendientes(n₁), n₁, descendientes(n₂), n₂, ..., descendientes(n_k), n_k, n).

ENTONCES: La raíz es el primero de oprev, voy al segundo de oprev y me fijo los elementos que le siguen inmediatamente. Me fijo en opost si están antes o después de él, si están antes son descendientes, si están después no lo son.

```

1. void lisp_print(tree &T, iterator n) {
2.   iterator c = /* hijo mas izquierdo de n ... */;
3.   if (/* 'c' es 'Lambda'... */) {
4.     cout << /* valor en el nodo 'n' ... */;
5.   } else {
6.     cout << "(" << /* valor de 'n' ... */;
7.     while (/* 'c' no es 'Lambda'... */) {
8.       cout << " ";
9.       lisp.print(T, c);
10.      c = /* hermano derecho de c ... */;
11.    }
12.    cout << ")";
13.  }
14. }
```

INSERCIÓN EN ÁRBOLES

- Cuando insertamos un nodo en una posición Λ , entonces el elemento pasa a generar un nuevo nodo en donde estaba el nodo ficticio Λ .
- Cuando insertamos un nodo en una posición dereferenciable, entonces simplemente el elemento pasa a generar un nuevo nodo hoja en el lugar en esa posición, tal como operaría la operación de inserción del TAD lista en la lista de hijos. Entonces, si tengo un nodo a cuyo hijo es un nodo b, e inserto c en la posición de b, entonces la lista de hijos pasa a ser (c, b) de manera que c pasa a ser el hijo más izquierdo de a.
- Así como en listas insert(p,x) invalida las posiciones después de p (inclusive), en el caso de árboles, una inserción en el nodo n invalida las posiciones que son descendientes de n y que están a la derecha de n.

```

1. iterator tree_copy(tree &T, iterator nt,
2.                     tree &Q, iterator nq) {
3.     nq = /* nodo resultante de insertar el
4.           elemento de 'nt' en 'nq' ... */;
5.     iterator
6.     ct = /* hijo mas izquierdo de 'nt' ... */,
7.     cq = /* hijo mas izquierdo de 'nq' ... */;
8.     while /* 'ct' no es 'Lambda'... */) {
9.         cq = tree_copy(T,ct,Q,cq);
10.        ct = /* hermano derecho de 'ct'... */;
11.        cq = /* hermano derecho de 'cq'... */;
12.    }
13.    return nq;
14. }
```

```

1. iterator mirror_copy(tree &T, iterator nt,
2.                      tree &Q, iterator nq) {
3.     nq = /* nodo resultante de insertar
4.           el elemento de 'nt' en 'nq' ... */;
5.     iterator
6.     ct = /* hijo mas izquierdo de 'nt' ... */,
7.     cq = /* hijo mas izquierdo de 'nq' ... */;
8.     while /* 'ct' no es 'Lambda'... */) {
9.         cq = mirror_copy(T,ct,Q,cq);
10.        ct = /* hermano derecho de 'ct' ... */;
11.    }
12.    return nq;
13. }
```

Algoritmos para copiar un árbol. El primero copia el subárbol del nodo nt en el árbol Q, y devuelve la posición de la raíz del subárbol insertado en Q. El segundo copia en espejo (nodos hermanos invertidos).

SUPRESIÓN EN ÁRBOLES

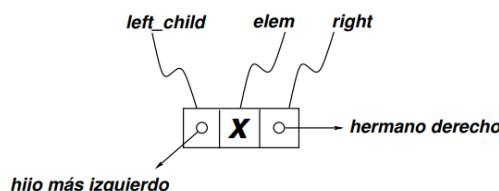
Solo se puede suprimir en posiciones dereferenciables. En el caso de suprimir en un nodo hoja, sólo se elimina el nodo. Si el nodo tiene hijos, eliminarlo equivale a eliminar todo el subárbol correspondiente. Como en listas, eliminando un nodo devuelve la posición del hermano derecho que llena el espacio dejado por el nodo eliminado.

```

1. class iterator_t {
2.     /* ... */
3. public:
4.     iterator_t lchild();
5.     iterator_t right();
6. };
7.
8. class tree {
9.     /* ... */
10. public:
11.     iterator_t begin();
12.     iterator_t end();
13.     elem_t &retrieve(iterator_t p);
14.     iterator_t insert(iterator_t p, elem_t t);
15.     iterator_t erase(iterator_t p);
16.     void clear();
17.     iterator_t splice(iterator_t to, iterator_t from);
18. };
```

INTERFAZ BÁSICA PARA ÁRBOLES (compatible con STL)

IMPLEMENTACIÓN POR PUNTEROS



En celdas, se almacenan el dato elem, un puntero right a la celda que corresponde al hermano derecho y otro left_child al hijo más izquierdo. El tipo iterator_t se define como un typedef a cell*.

Se mantiene el criterio de usar posiciones adelantadas con respecto al dato. Luego, el iterator consiste en tres punteros a celdas: un puntero ptr a la celda que contiene el dato (nulo en el caso de posiciones no dereferenciables), un puntero prev al hermano izquierdo (puede ser nulo), un puntero father al padre (nunca es nulo).

La raíz del árbol, si existe, es una celda hija de la celda de encabezamiento.

TIEMPOS DE EJECUCIÓN (INTERFAZ AVANZADA)

Operación	$T(n)$
begin(), end(), n.right(), n++, n.lchild(), *n, insert(), splice(to, from)	$O(1)$
erase(), find(), clear(), T1=T2	$O(n)$

Todas las funciones básicas tienen costo $O(1)$ porque la operación de mover todo el árbol de una posición a otra se realiza con una operación de punteros.

Las operaciones que no son $O(1)$ son erase(p) que debe eliminar todos los nodos del subárbol del nodo p, clear() que equivale a erase(begin()), find(x) y el constructor por copia ($T1=T$). En todos los casos, n es o bien el número de nodos del subárbol (erase(p) y find(x,p)) o bien el número total de nodos del árbol (clear(), find(x) y el constructor por copia $T1=T2$).

ÁRBOLES BINARIOS

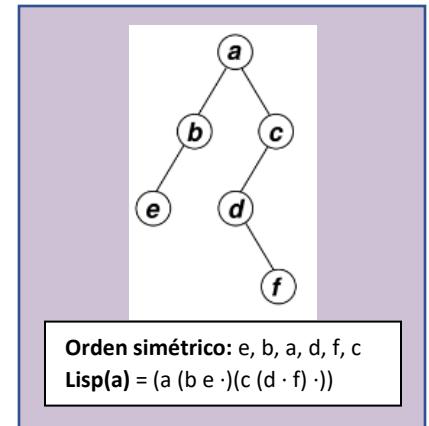
Un **árbol binario (btree)** tiene nodos que pueden tener, a lo sumo, dos hijos. Además, si un dato nodo n tiene un solo hijo, entonces este puede ser el hijo derecho o el hijo izquierdo de n . Es decir, un árbol binario con un nodo A con hijo izquierdo B es distinto a un árbol binario con un nodo A con hijo derecho B .

- Listado en orden simétrico:**

$\text{osim}(\Lambda) = \langle \text{lista vacía} \rangle$
 $\text{osim}(n) = (\text{osim}(s_l), n, \text{osim}(s_r))$ donde $s_{l,r}$ son los hijos izquierdo y derecho.

- Notación lisp:** Si un nodo tiene un solo hijo, se pone un punto en la posición del hijo faltante.

$$\text{lisp}(n) = \begin{cases} n & ; \text{ si } s_l = \Lambda \text{ y } s_r = \Lambda \\ (n \text{ lisp}(s_l) \text{ lisp}(s_r)) & ; \text{ si } s_l \neq \Lambda \text{ y } s_r \neq \Lambda \\ (n \cdot \text{lisp}(s_r)) & ; \text{ si } s_r \neq \Lambda \\ (n \text{ lisp}(s_l) \cdot) & ; \text{ si } s_l \neq \Lambda \end{cases}$$



NODOS EN UN ÁRBOL BINARIO: Un árbol binario puede tener a lo sumo 2^L nodos en el nivel L . Luego, un árbol binario de L niveles puede tener a lo sumo $n \leq 1 + 2 + 4 + \dots + 2^L$ nodos. Esto es una serie geométrica de razón dos, tal que $n < 2^{L+1}$.

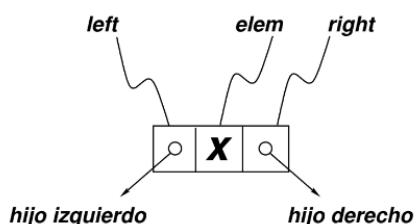
A partir de los niveles se puede obtener el número de nodos como $l \geq \text{floor}(\log_2 n)$.

ÁRBOLES IGUALES: Ambos son vacíos o ambos son no vacíos, los valores de sus nodos son iguales y los hijos respectivos de su nodo raíz son iguales.

ÁRBOLES SEMEJANTES: Ambos son vacíos o ambos son no vacíos, y los hijos respectivos de su nodo raíz son semejantes (no se tiene en cuenta el valor de nodo).

OPERACIONES BÁSICAS:

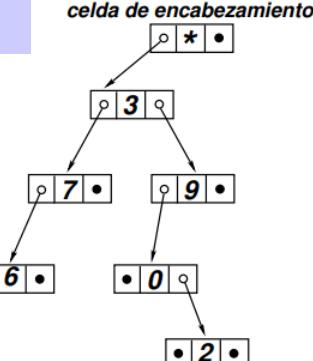
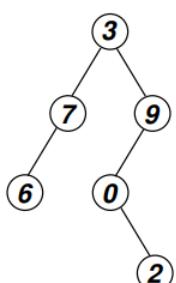
- Dado un nodo, obtener su hijo izquierdo o su hijo derecho (puede retornar una posición Λ).
- Dada una posición, determinar si es Λ o no.
- Obtener la posición de la raíz del árbol.
- Dado un nodo obtener una referencia al dato contenido en el nodo.
- Dada una posición no dereferenciable y un dato, insertar un nuevo nodo con ese dato en esa posición.
- Borrar un nodo y todo su subárbol correspondiente.



```
1. class cell {
2.   friend class btreet;
3.   friend class iterator_t;
4.   elem_t t;
5.   cell *right,*left;
6.   cell() : right(NULL), left(NULL) {}
7. };
```

```
1. class iterator_t {
2. private:
3.   friend class btreet;
4.   cell *ptr,*father;
5.   enum side_t {NONE,R,L};
6.   side_t side;
7.   iterator_t(cell *p,side_t side_a,cell *f_a)
8.   : ptr(p), side(side_a), father(f_a) {}
```

IMPLEMENTACIÓN POR PUNTEROS



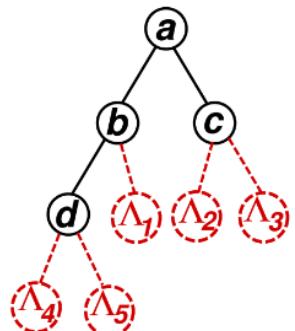
PROGRAMACION FUNCIONAL: Programación en los cuales los datos de los algoritmos pueden ser también funciones. Esto es: $T(*f)(T)$ ó $\text{bool } (*\text{pred})(\text{int } x)$ ó $[](\text{int } x)$.

Contiene un puntero a la celda y otro al padre, como en el caso del AOO.

El puntero prev que apunta al hermano a la izquierda, aquí ya no tiene sentido. Recordemos que el iterator nos debe permitir ubicar a las posiciones, incluso aquellas que son Λ . Para ello, incluimos en el iterator un miembro side de tipo enum side_t que puede tomar los valores R (right) y L (left).

- nodo b : $\text{ptr}=b, \text{father}=a, \text{side}=L$
- nodo c : $\text{ptr}=c, \text{father}=a, \text{side}=R$
- nodo d : $\text{ptr}=d, \text{father}=b, \text{side}=L$
- nodo Λ_1 : $\text{ptr}=\Lambda, \text{father}=b, \text{side}=R$
- nodo Λ_2 : $\text{ptr}=\Lambda, \text{father}=c, \text{side}=L$
- nodo Λ_3 : $\text{ptr}=\Lambda, \text{father}=c, \text{side}=R$
- nodo Λ_4 : $\text{ptr}=\Lambda, \text{father}=d, \text{side}=L$
- nodo Λ_5 : $\text{ptr}=\Lambda, \text{father}=d, \text{side}=R$

Notar que si no tuviéramos el campo side, Λ_4 y Λ_5 serían iguales.



ÁRBOLES DE HUFFMAN

Un **árbol de Huffman** es un tipo de árbol binario que se utiliza para comprimir archivos o mensajes de texto, buscando una representación del mensaje en bits (cadenas de 0's y 1's) lo más corta posible. El algoritmo debe permitir recuperar el mensaje original ya que, de esta forma, si la cadena de caracteres representa un archivo, entonces podemos guardar el mensaje codificado (que es más corto) con el consecuente ahorro de espacio.

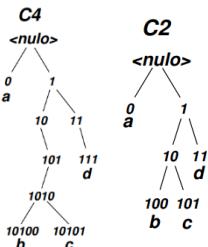
- Si se utiliza la representación binaria ASCII de los caracteres, cada carácter se encoda en un código de 8 bits. Si el mensaje tiene N caracteres, entonces el mensaje codificado tendrá una longitud de $8N$ bits, resultando en una longitud promedio de $\langle l \rangle = l/N$ (8 bits/carácter). En general, si el número de caracteres es n_c , la tasa será de $\langle l \rangle = \text{ceil}(\log_2 n_c)$. Esto es el código C1.
- Utilizando códigos de longitud variable, se asignan códigos de longitud corta a los caracteres que tienen más probabilidad de aparecer y códigos más largos a los que tienen menos probabilidad de aparecer. Esto es el código C2.
- Condición de prefijos:** En el caso de que un código de longitud variable como el C2 sea más conveniente, debemos poder asegurarnos de poder desencodar los mensajes, es decir, que la relación entre mensaje y mensaje codificado sea única y que exista un algoritmo para encodar y desencodar mensajes en un tiempo razonable. No sólo los códigos de los diferentes caracteres deben ser diferentes entre sí, sino que también se necesita que se cumpla la condición de prefijos: *que el código de un carácter no sea prefijo del código de ningún otro carácter*.

Letra	Código C1	Código C2	Código C3
a	00	0	0
b	01	100	01
c	10	101	10
d	11	11	101

En C3, el código de c es prefijo de d, luego no cumple con la condición de prefijos.

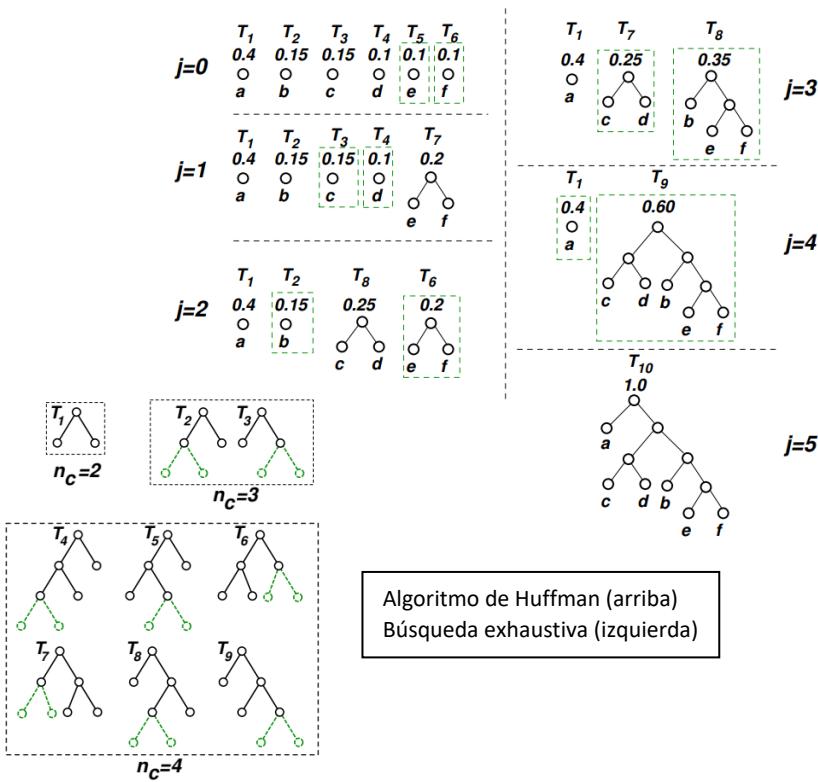
CODIFICACIÓN DE UN ÁRBOL DE HUFFMAN: a la raíz se le asocia un código de longitud 0 <nulo>, luego la definición es recursiva tal que si un nodo tiene código $b_0b_1\dots b_{n-1}$ entonces el hijo izquierdo tiene código $b_0b_1\dots b_{n-1}0$ y el hijo derecho $b_0b_1\dots b_{n-1}1$. En el nivel I se encuentran todos los códigos de longitud l. En este caso, para que se cumpla la condición de prefijos se exige que los caracteres estén sólo en las hojas, no en los nodos interiores.

Si tenemos nodos interiores con un solo hijo, el código es redundante ya que se pueden “subir” los subárboles de las hojas y obtener una codificación menor. Por ejemplo:



La codificación de abcd para C4 es 0101001010111, mientras que para C2 es 010010111, haciéndolo significativamente más simple.

A C2 se le llama **árbol binario lleno (FBT)**, que es cuando en un árbol binario los nodos son o bien hojas, o bien nodos interiores con sus dos hijos.



ALGORITMO DE HUFFMAN:

- Se crean n_c árboles (tantos caracteres como hay) con una sola hoja asociada a uno de los caracteres. Se le asocia un peso total, inicialmente le de cada carácter.
- En sucesivas iteraciones el algoritmo va combinando los dos árboles con menor peso en uno sólo. Como en cada combinación desaparecen dos árboles y aparece uno nuevo, en $n_c - 1$ iteraciones queda un solo árbol con el código resultante.

Cuando varios árboles tienen la misma probabilidad, se puede tomar cualquiera dos de ellos, no importa si se pone un subárbol a la izquierda o derecha para la longitud media.

La **longitud media** se calcula como la suma de los pesos por la longitud, tal que $\langle l \rangle = \sum P(c)l(c)$. La longitud es lo mismo que la profundidad del nodo en el árbol resultante del algoritmo.

A diferencia de los algoritmos heurísticos, se llega a la tabla de códigos óptima y la longitud media por carácter coincide con la de estos pero a un costo mucho menor.

BUSQUEDA EXHAUSTIVA:

Se generan todos los posibles árboles llenos de n_c caracteres, se calcula la longitud promedio de cada uno de ellos y nos quedamos con el que tiene la longitud promedio mínima.

Los distintos árboles se arman agregando dos hojas en cada una de las hojas del árbol anterior.

Como el número de permutaciones de n_c caracteres es $n_c!$, tenemos que en total hay a lo sumo $(n_c - 1)!n_c!$ posibles tablas de códigos. Luego, $T(n_c) = (n_c - 1)!n_c! < (n_c!)^2 = O(n_c^{2n_c})$. Esto es, se descarta esta estrategia para mensajes con $n_c > 20$.

CONJUNTOS

Un **conjunto** es una colección de “miembros” o “elementos” de un “conjunto universal”. Todos los miembros del conjunto deben ser diferentes, es decir no puede haber dos copias del mismo elemento. No es lo mismo que una lista, ya que no existe un orden preestablecido entre los miembros de un conjunto.

Para definir el concepto de conjunto sólo es necesario el concepto de igualdad o desigualdad entre los elementos del conjunto universal, pero en general las representaciones de conjuntos asumen que entre ellos existe además una **relación de orden estricta**, que usualmente se denota como <.

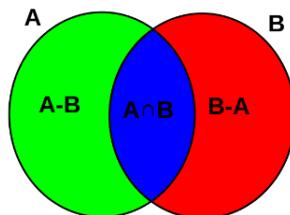
- **NOTACIÓN:** Un conjunto se escribe enumerando sus elementos entre llaves, como {1, 4}, o con una condición sobre los miembros del conjunto universal, por ejemplo: $A = \{x \text{ entero} / x \text{ es par}\}$.

• RELACIONES DE CONJUNTOS:

- La pertenencia es la principal relación de los conjuntos, tal que $x \in A$ si x es un miembro de A .
- La inclusión, tal que A está incluido en B ($A \subseteq B$) si todo miembro de A también es miembro de B . Luego, A es un subconjunto de B y B es un supraconjunto de A . Todo conjunto está incluido en sí mismo y el conjunto vacío \emptyset está incluido en cualquier conjunto. Dos conjuntos A y B son iguales si $A \subseteq B$ y $B \subseteq A$, por lo tanto, dos conjuntos son distintos si al menos existe un elemento de A que no pertenece a B o viceversa.
- El conjunto A es un subconjunto propio (supraconjunto propio) de B si $A \subseteq B$ ($A \neq B$) y $A \neq B$.

• RELACIONES DE CONJUNTOS:

- La unión $A \cup B$ es el conjunto de los elementos que pertenecen a A y a B .
- La intersección $A \cap B$ es el conjunto de los elementos que pertenecen tanto a A como a B .
- La diferencia $A - B$ está formada por los elementos de A que no están en B .
- $A \cup B = (A \cap B) \cup (A - B) \cup (B - A)$, siendo los tres conjuntos del miembro derecho disjuntos.



```
1. typedef int elem_t;
2.
3. class iterator_t {
4. private:
5.   /* ... */;
6. public:
7.   bool operator!=(iterator_t q);
8.   bool operator==(iterator_t q);
9. };
10.
11. class set {
12. private:
13.   /* ... */;
14. public:
15.   set();
16.   set(const set &);
17.   ~set();
18.   elem_t retrieve(iterator_t p);
19.   pair<iterator_t, bool> insert(elem_t t);
20.   void erase(iterator_t p);
21.   int erase(elem_t x);
22.   void clear();
23.   iterator_t next(iterator_t p);
24.   iterator_t find(elem_t x);
25.   iterator_t begin();
26.   iterator_t end();
27. };
28. void set_union(set &A, set &B, set &C);
29. void set_intersection(set &A, set &B, set &C);
30. void set_difference(set &A, set &B, set &C);
```

INTERFAZ BÁSICA PARA CONJUNTOS:

- Una clase **iterator** permite recorrer el contenedor. Los iterators soportan los operadores comparación == y !=. begin(), end() y next() permiten iterar sobre el conjunto.
- En el conjunto NO se puede insertar un elemento en una posición determinada, por lo tanto, la función **insert** no tiene un argumento posición como en listas o árboles. insert(x) retorna un pair<iterator,bool>. El primero es, como siempre, un iterator al elemento insertado. El segundo indica si el elemento realmente fue insertado o ya estaba en el conjunto.
- La función **erase(p)** elimina el elemento que está en la posición p . La posición p debe ser válida y dereferenciable, es decir debe haber sido obtenida de un insert(x) o find(x). erase(p) invalida p y todas las otras posiciones obtenidas previamente.
- **erase(x)** elimina el elemento si x estaba en el conjunto. Si no, el conjunto queda inalterado, devuelve el número de elementos efectivamente eliminados (puede ser 0 o 1). Debería ser un bool pero por compatibilidad con multiset es un int (en un multiset pueden haber varias copias de un mismo elemento, entonces puede ser más de 1 elementos eliminados).
- $p = \text{find}(x)$ devuelve un iterator a la posición ocupada por el elemento x en el conjunto. Si el conjunto no contiene a x , entonces devuelve end().
- Las operaciones binarias sobre conjuntos se realizan con las funciones **set_union(A,B,C)**, **set_intersection(A,B,C)** y **set_difference(A,B,C)** que corresponden a las operaciones $C = A \cup B$, $C = A \cap B$ y $C = A - B$, respectivamente. En todas las funciones binarias, ninguno de los conjuntos de entrada (ni A ni B) debe superponerse (overlap) con C : es decir, que si queremos dejar en A sólo los elementos que están también en B no puedo hacer **set_intersection(A,B,A)**, tengo que hacer **set_difference(A,B,tmp)** y después $A = \text{tmp}$.

IMPLEMENTACIÓN POR VECTORES DE BITS:

La forma más simple de representar un conjunto es guardando un campo de tipo bool por cada elemento del conjunto universal. Si este campo es verdadero, entonces el elemento está en el conjunto y viceversa. Por ejemplo, si el conjunto universal son los enteros de 0 a $N - 1$, $U = \{j \text{ entero, tal que } 0 \leq j < N\}$.

Entonces podemos representar a los conjuntos por vectores de valores booleanos (por ejemplo, **vector<bool>**) de longitud N . Si v es el vector, si el conjunto es $S = \{4, 6, 9\}$ y N es 10, entonces el vector de bits correspondiente sería $v = \{0, 0, 0, 1, 0, 1, 0, 0, 1\}$.

Para insertar o borrar elementos se prende o apaga el bit correspondiente. Todas estas operaciones son O(1).

Las operaciones binarias también son muy simples de implementar: si quiero hacer la unión $C = A \cup B$, entonces debemos hacer $C.v[j] = A.v[j] \mid\mid B.v[j]$. La intersección se obtiene con $C.v[j] = A.v[j] \&\& B.v[j]$, y la diferencia $C = A - B$ con $C.v[j] = A.v[j] \&\& !B.v[j]$. El tiempo de ejecución de estas operaciones es $O(N)$, donde N es el número de elementos del conjunto universal. La memoria requerida es N bits, es decir que también es $O(N)$. Para **vector<int>** o **vector<char>** también es $O(N)$ al ser $N * \text{sizeof}(T)$.

FUNCIONES AUXILIARES PARA DEFINIR CONJUNTOS

Para representar conjuntos universales U que no son subconjuntos de los enteros, o que no son un subconjunto continuo de los enteros [0,N] debemos definir funciones que establezcan la correspondencia entre los elementos del conjunto universal y el conjunto de los enteros entre [0,N]. Las llamamos **funciones auxiliares para definir conjuntos** dentro de un determinado conjunto universal U.

idx() determina el rango y la función element(), en forma recíproca, devuelve el valor del elemento.

```
1. int indx(elem_t t);
2. elem_t element(int j);
```

Declaración de las funciones auxiliares.

```
1. const int N=50;
2. typedef int elem_t;
3. int indx(elem_t t) { return (t-100)/2; }
4. elem_t element(int j) { return 100+2*j; }
```

Implementaciones para números pares entre 100 y 198, y letras tanto en minúsculas como en mayúsculas.

```
1. const int N=52;
2. typedef char elem_t;
3. int indx(elem_t c) {
4.     if (c>='a' && c<='z') return c-'a';
5.     else if (c>='A' && c<='Z') return 26+c-'A';
6.     else cout << "Elemento fuera de rango!!\n"; abort();
7. }
8. elem_t element(int j) {
9.     assert(j<N);
10.    return (j<26 ? 'a'+j : 'A'+j-26);
11. }
```

DESCRIPCIÓN DEL CÓDIGO CON VECTORES

La clase **iterator** es simplemente un **typedef** de los enteros, con la particularidad de que el iterator corresponde al índice en el conjunto universal, es decir el valor que retorna la función **indx()**. El iterator correspondiente a 100 es 0, a 102 es 1 y así hasta 198 que le corresponde 50.

Se elige como **end()** el índice N, que nunca será usado por un elemento del conjunto. Los iterators sólo deben avanzar sobre los valores definidos en el conjunto (si tengo S = {100,128,180}, p = S.begin() debe retornar 10 y aplicar p = S.next(p) devuelve 14, 40 y 50 sucesivamente).

p = next_aux(p) devuelve el primer índice siguiente a p ocupado por un elemento. La función **next()** incrementa p y luego le aplica **next_aux()** para avanzar hasta el siguiente elemento del conjunto.

La función **retrieve(p)** devuelve el elemento usando la función **element(p)**. **erase(x)** e **insert(x)** simplemente prenden o apagan la posición correspondiente **v[indx(x)]**. Notar que son O(1).

find(x) verifica si el elemento está en el conjunto, en ese caso retorna **indx(x)**, sino retorna N (que es **end()**).

size() cuenta las posiciones en v que están prendidas. Por lo tanto es O(N).

Las funciones binarias **set_union**, **set_intersection** y **set_difference** hacen un lazo sobre todas las posiciones del vector y por lo tanto son O(N).

IMPLEMENTACIÓN CON LISTAS

El conjunto es muy cercano a la correspondencia y se puede representar conjuntos con correspondencias, usando las claves de la correspondencia como elementos del conjunto e ignorando el valor del contradominio.

TAD conjunto	TAD Correspondencia
x = retrieve(p);	x = retrieve(p).first;
p=insert(x)	p=insert(x,w)
erase(p)	erase(p)
erase(x)	erase(x)
clear()	clear()
p = find(x)	p = find(x)
begin()	begin()
end()	end()

Solo falta implementar las funciones binarias. Con contenedores no ordenados, **set_intersection(A,B,C)** es $O(n_A * n_B)$ tal que si el número de elementos en los dos contenedores es similar, es $O(n^2)$.

Con contenedores ordenados se puede disminuir el tiempo de ejecución (orden) de las funciones binarias.

```
1. typedef int iterator_t;
2.
3. class set {
4. private:
5.     vector<bool> v;
6.     iterator_t next_aux(iterator_t p) {
7.         while (p<N && !v[p]) p++;
8.         return p;
9.     }
10. public:
11.     set() : v(N,0) {}
12.     set(const set &A) : v(A.v) {}
13.     ~set() {}
14.     iterator_t lower_bound(elem_t x) {
15.         return next_aux(indx(x));
16.     }
17.     pair_t insert(elem_t x) {
18.         iterator_t k = indx(x);
19.         bool inserted = !v[k];
20.         v[k] = true;
21.         return pair_t(k,inserted);
22.     }
23.     elem_t retrieve(iterator_t p) { return element(p); }
24.     void erase(iterator_t p) { v[p]=false; }
25.     int erase(elem_t x) {
26.         iterator_t p = indx(x);
27.         int r = (v[p] ? 1 : 0);
28.         v[p] = false;
29.         return r;
30.     }
31.     void clear() { for(int j=0; j<N; j++) v[j]=false; }
32.     iterator_t find(elem_t x) {
33.         int k = indx(x);
34.         return (v[k] ? k : N);
35.     }
36.     iterator_t begin() { return next_aux(0); }
37.     iterator_t end() { return N; }
38.     iterator_t next(iterator_t p) { next_aux(++p); }
39.     int size() {
40.         int count=0;
41.         for (int j=0; j<N; j++) if (v[j]) count++;
42.         return count;
43.     }
44.     friend void set_union(set &A,set &B,set &C);
45.     friend void set_intersection(set &A,set &B,set &C) {
46.         for (int j=0; j<N; j++) C.v[j] = A.v[j] && B.v[j];
47.     }
48.     friend void set_difference(set &A,set &B,set &C);
49.     void set_union(set &A,set &B,set &C) {
50.         for (int j=0; j<N; j++) C.v[j] = A.v[j] | | B.v[j];
51.     }
52.     void set_intersection(set &A,set &B,set &C) {
53.         for (int j=0; j<N; j++) C.v[j] = A.v[j] && B.v[j];
54.     }
55.     void set_difference(set &A,set &B,set &C) {
56.         for (int j=0; j<N; j++) C.v[j] = A.v[j] && ! B.v[j];
57.     }
58. }
```

FUNCIONES BINARIAS CON LISTAS ORDENADAS: Para cualquiera de las operaciones binarias inicializamos los iterators con $pa = A.begin()$ y $pb = B.begin()$ y avanzamos el menor o, si son iguales, ambos. El proceso se detiene cuando alguno de los iterators avanza, de manera que es claro que en un número finito de pasos alguno de los iterators llegará al fin, de hecho, en menos de $n_a + n_b$ pasos (luego, $O(n_a + n_b)$).

Se recorren todos los elementos de alguno de los dos conjuntos, mientras que en el otro puede quedar un “resto”, es decir, una cierta cantidad de elementos al final de la lista.

UNION	INTERSECCIÓN	DIFERENCIA
<ul style="list-style-type: none"> $x_a=1, x_b=3$, inserta 1 en C $x_a=3, x_b=3$, inserta 3 en C $x_a=5, x_b=5$, inserta 5 en C $x_a=7, x_b=7$, inserta 7 en C $x_a=10, x_b=9$, inserta 9 en C $x_a=10, x_b=10$, inserta 10 en C pa llega a $A.end()$ Inserta todo el resto de B (el elemento 12) en C. 	<ul style="list-style-type: none"> $x_a=1, x_b=3$, inserta 1 en C $x_a=3, x_b=3$, inserta 3 en C $x_a=5, x_b=5$, inserta 5 en C $x_a=7, x_b=7$, inserta 7 en C $x_a=10, x_b=9$, inserta 9 en C $x_a=10, x_b=10$, inserta 10 en C pa llega a $A.end()$ <p>Sólo insertamos x_a cuando $x_a = x_b$ y los restos no se insertan.</p>	<ul style="list-style-type: none"> $x_a=1, x_b=3$, inserta 1 $x_a=3, x_b=3$, $x_a=5, x_b=5$, $x_a=7, x_b=7$, $x_a=10, x_b=9$, $x_a=10, x_b=10$, pa llega a $A.end()$ <p>Insertar si $x_a < x_b$ y x_b no se inserta nunca. Al final se inserta el resto de A.</p>

```
void set_union(set &A, set &B, set &C) {
    C.clear();
    list<elem_t>::iterator pa = A.L.begin(),
        pb = B.L.begin(), pc = C.L.begin();
    while (pa!=A.L.end() && pb!=B.L.end()) {
        if (*pa<*pb) { pc = C.L.insert(pc,*pa); pa++; }
        else if (*pa>*pb) { pc = C.L.insert(pc,*pb); pb++; }
        else { pc = C.L.insert(pc,*pa); pa++; pb++; }
        pc++;
    }
    while (pa!=A.L.end()) {
        pc = C.L.insert(pc,*pa);
        pa++; pc++;
    }
    while (pb!=B.L.end()) {
        pc = C.L.insert(pc,*pb);
        pb++; pc++;
    }
}
```

```
void set_intersection(set &A, set &B, set &C) {
    C.clear();
    list<elem_t>::iterator pa = A.L.begin(),
        pb = B.L.begin(), pc = C.L.begin();
    while (pa!=A.L.end() && pb!=B.L.end()) {
        if (*pa<*pb) pa++;
        else if (*pa>*pb) pb++;
        else { pc = C.L.insert(pc,*pa); pa++; pb++; pc++; }
    }
}
```

```
void set_difference(set &A, set &B, set &C) {
    C.clear();
    list<elem_t>::iterator pa = A.L.begin(),
        pb = B.L.begin(), pc = C.L.begin();
    while (pa!=A.L.end() && pb!=B.L.end()) {
        if (*pa<*pb) { pc = C.L.insert(pc,*pa); pa++; pc++; }
        else if (*pa>*pb) pb++;
        else { pa++; pb++; }
    }
    while (pa!=A.L.end()) {
        pc = C.L.insert(pc,*pa);
        pa++; pc++;
    }
}
```

```
typedef int elem_t;
typedef list<elem_t>::iterator iterator_t;

class set {
private:
    list<elem_t> L;
public:
    set() {}
    set(const set &A) : L(A.L) {}
    ~set() {}
    elem_t retrieve(iterator_t p) { return *p; }
    iterator_t lower_bound(elem_t t) {
        list<elem_t>::iterator p = L.begin();
        while (p!=L.end() && t>*p) p++;
        return p;
    }
    iterator_t next(iterator_t p) { return ++p; }
    pair<iterator_t,bool> insert(elem_t x) {
        pair<iterator_t,bool> q;
        iterator_t p;
        p = lower_bound(x);
        q.second = p==end() || *p!=x;
        if(q.second) p = L.insert(p,x);
        q.first = p;
        return q;
    }
    void erase(iterator_t p) { L.erase(p); }
    void erase(elem_t x) {
        list<elem_t>::iterator
            p = lower_bound(x);
        if (p!=end() && *p==x) L.erase(p);
    }
    void clear() { L.clear(); }
    iterator_t find(elem_t x) {
        list<elem_t>::iterator
            p = lower_bound(x);
        if (p!=end() && *p==x) return p;
        else return L.end();
    }
    iterator_t begin() { return L.begin(); }
    iterator_t end() { return L.end(); }
    int size() { return L.size(); }
    friend void set_union(set &A, set &B, set &C);
    friend void set_intersection(set &A, set &B, set &C);
    friend void set_difference(set &A, set &B, set &C);
};
```

Método	$T(N)$
retrieve(p), insert(x), erase(x), clear(), find(x), lower_bound(x), set_union(A, B, C), set_intersection(A, B, C), set_difference(A, B, C),	$O(n)$
erase(p), begin(), end(),	$O(1)$

Tiempos de ejecución del TAD conjunto implementado con listas ordenadas.

INTERFAZ AVANZADA PARA CONJUNTOS

La clase set pasa a ser un template, de manera que se pueda declarar $\text{set}<\text{int}>$, $\text{set}<\text{double}>$.

La clase iterator es ahora una clase anidada dentro de set. Externamente se verá como $\text{set}<\text{int}>::\text{iterator}$.

La dereferenciación de posiciones ($x = \text{retrieve}(p)$) se reemplaza por $x = *p$ sobrecargando el operador *. Si el tipo elemento (es decir el tipo T del template) contiene campos dato o métodos, podemos escribir $p->\text{campo}$ o $p->\text{f}(...)$. Para esto sobrecargamos los operadores operator* y operator->.

Igual que con la interfaz básica, para poder hacer comparaciones de iterators debemos sobrecargar también los operadores == y != en la clase iterator.

erase(x) retorna el número de elementos efectivamente eliminados.

insert(x) retorna un $\text{pair}<\text{iterator}, \text{bool}>$. El primero es, como siempre, un iterator al elemento insertado. El segundo indica si el elemento realmente fue insertado o ya estaba en el conjunto.

EL DICCIONARIO

Es un TAD conjunto sin necesidad de las funciones binarias. Existe una implementación muy eficiente para la cual las inserciones y supresiones son $O(1)$, basada en la estructura “**tabla de dispersión**” (“hash tables”). Sin embargo, no es simple implementar en forma eficientes las operaciones binarias para esta implementación, por lo cual no es buen candidato para conjuntos, pero sí se usa para el TAD diccionario.

TABLA DE DISPERSIÓN

La idea esencial es que dividimos el conjunto universal en B **cubetas** (“buckets” o “bins”) de tal manera que, a medida que nuevos elementos son insertados en el diccionario, estos son desviados a la cubeta correspondiente. En general tendremos una **función de dispersión** $\text{int } b = h(\text{elem_t } t)$ que nos da el número de cubeta b en el cual debe ser almacenado el elemento t .

FUNCIONAMIENTO: Básicamente, las cubetas son guardadas en un arreglo de cubetas (vector<elem_t> v(B)). Para insertar un elemento, calculamos la cubeta usando la función de dispersión y guardamos el elemento en esa cubeta. Para hacer un `find(x)` o `erase(x)`, calculamos la cubeta y verificamos si el elemento está en la cubeta o no, luego, estas operaciones son $O(1)$.

DESVANTAJA: Pero normalmente el número de cubetas es mucho menor que el número de elementos del conjunto universal N (en muchos casos este es infinito). Si un elemento es insertado y la cubeta correspondiente ya está ocupada, hay una **colisión** y no podemos insertar el elemento en la tabla. Si es el mismo elemento que estamos intentando insertar (un 1 en el lugar de un 1) no pasa nada, pero si es un 24 en lugar de un 4, hay un problema.

- **EFFECTO DE CUMPLEAÑOS:** Si el número de elementos en el conjunto n es pequeño con respecto al número de cubetas, entonces la probabilidad de que dos elementos vayan a la misma cubeta es pequeña, pero en este caso la memoria requerida (que es el tamaño del vector v , es decir $O(B)$) sería mucho mayor que el tamaño del conjunto, con lo cual deja de tener la misma utilidad práctica. Si tenemos B cubetas y generamos m objetos aleatorios, la probabilidad de que al menos 2 de ellos caigan en la misma cubeta es del 50% cuando $m = \sqrt{B}$.

Luego, se tienen dos estrategias para resolver colisiones, las **tablas de dispersión abiertas** y la redispersión en **tablas de dispersión cerradas**:

TABLAS DE DISPERSIÓN ABIERTAS:

En esta implementación las cubetas no son elementos, sino que son listas (simplemente enlazadas) de elementos, es decir, el vector v es de tipo `vector<list<elem_t>>`. De esta forma, la cubeta puede contener (teóricamente) infinitos elementos. Los elementos pueden insertarse en las cubetas de forma ordenada o desordenada, lo cual difiere en eficiencia.

- **Listas desordenadas:** La inserción de un elemento x pasa por calcular el número de cubeta usando la función de dispersión y revisar la lista (cubeta) correspondiente. Si el elemento está en la lista, entonces no es necesario hacer nada. Si no está, podemos insertar el elemento en cualquier posición, puede ser en `end()`.

El costo de la inserción, en el peor caso, es cuando el elemento no está en la lista, proporcional al número de elementos en la lista (cubeta).

Si tenemos n elementos, el número de elementos por cubeta será, en promedio, n/B . Si el número de cubetas es $B \approx n$, entonces $n/B \approx 1$ y el tiempo de ejecución es $O(1 + n/B)$. El 1 tiene en cuenta acá de que al menos hay que calcular la función de dispersión. En el peor caso, todos los elementos pueden terminar en una sola cubeta, en cuyo caso la inserción sería $O(n)$. Algo similar ocurre con `erase`.

Método	$T(n, B)$ (promedio)	$T(n, B)$ (peor caso)
<code>retrieve(p)</code> , <code>erase(p)</code> , <code>end()</code>	$O(1)$	$O(1)$
<code>insert(x)</code> , <code>find(x)</code> , <code>erase(x)</code>	$O(1 + n/B)$	$O(n)$
<code>begin()</code> , <code>next(p)</code>	$\begin{cases} \text{si } n = 0, & O(B); \\ \text{si } n \leq B, & O(B/n); \\ \text{si } n > B, & O(1); \end{cases}$	$O(B)$
<code>clear()</code>	$O(n + B)$	$O(n + B)$

FUNCIONES DE DISPERSIÓN:

El diseño de una buena función de dispersión implica que los elementos sean distribuidos uniformemente sobre las cubetas. Se deben considerar ensambles de elementos representativos y ver que tan bien se desempeñan las funciones potenciales para esos ensambles.

IMPLEMENTACIÓN

- La clase contiene un puntero h a la función de dispersión. El constructor toma como argumentos el número de cubetas y el puntero a la función de dispersión y los copia en valores internos. Redimensiona el vector de cubetas v e inicializa el contador de elementos `count`.
- La clase iterator consiste de el número de cubeta (bucket) y un iterator en la lista (p). Las posiciones válidas son posiciones dereferenciable y `end()`.
- El iterator `end()` consiste en el par $bucket = B - 1$ y p el `end()` de esa lista (es decir, $v[bucket].end()$).
- Hay que tener cuidado de, al avanzar un iterator siempre llegar a otro iterator válido. La función privada `next_aux()` avanza cualquier combinación de cubeta y posición en la lista (puede no ser válida, por ejemplo el `end()` de una lista que no es la última) hasta la siguiente posición válida.
- El tiempo de ejecución de `next_aux()` es 1 si avanza una posición en la misma cubeta sin llegar al `end()`. Si esto último ocurre, entonces entra en un lazo sobre las cubetas del cual sólo sale cuando llega a una cubeta no vacía. Si el número de elementos es mayor que B entonces en promedio todas las cubetas tienen al menos un elemento y `next_aux()` a lo sumo debe avanzar a la siguiente cubeta. Es decir, el lazo en `next_aux()` se ejecuta una sola vez. Por cada cubeta llena hay B/n cubetas vacías y el lazo en `next_aux()` se ejecuta B/n veces.

TABLAS DE DISPERSIÓN CERRADAS:

Se utiliza otra cubeta cercana a la que indica la función de dispersión. A esto se le llama “**redispersión**”: la tabla es un vector `<key_t>` e inicialmente todos los elementos están inicializados a un valor especial que llamaremos `undef` (“indefinido”). Si el diccionario guarda valores enteros positivos, podemos usar 0 como `undef`. Si los valores son reales (double o float) podemos usar como `undef` los valores DBL_MAX o NAN. Si estamos almacenando nombres, podemos usar la cadena `<NONE>` y así siguiendo.

Para insertar un elemento x calculamos la función de dispersión $init = h(x) \% B$ para ver cuál cubeta le corresponde inicialmente. Si la cubeta está libre (es decir el valor almacenado es `undef`), entonces podemos insertar el elemento en esa posición. Si está ocupado, entonces podemos probar en la siguiente cubeta (en “sentido circular”, es decir si la cubeta es $B - 1$ la siguiente es la 0) ($init + 1) \% B$). Si está libre lo insertamos allí, si está ocupada y el elemento almacenado no es x , seguimos con la siguiente, etc... hasta llegar a una libre. Si todas las cubetas están ocupadas, la tabla está llena y el programa debe señalar un error (o agrandar la tabla dinámicamente).

Para hacer $find(x)$ no basta con buscar en la cubeta $h(x)$ porque el algoritmo puede haberlo insertado en alguna cubeta posterior a esa, si es que estaban llenas las otras. Luego, revisamos las cubetas a partir de $h(x)$, entonces cuando lleguemos a alguna cubeta que contiene a x podemos devolver el iterator correspondiente. Si llegamos a `undef`, sabemos que el elemento “no está” en el diccionario, ya que si estuviera debería haber sido insertado en alguna cubeta entre $h(x)$ y el siguiente `undef`.

TASA DE OCUPACIÓN $\alpha = n/B$:

COSTO DE LA INSERCIÓN EXITOSA:

El costo de insertar un nuevo elemento (**inserción exitosa**) es proporcional al número m de cubetas ocupadas que hay que recorrer hasta encontrar una cubeta libre. Cuando hay muy pocas cubetas ocupadas (α es mucho menor que 1), la probabilidad de encontrar una cubeta libre es grande y el costo de inserción es $O(1)$. A medida que la tabla se va llenando, la probabilidad es alta y el costo de inserción ya no es $O(1)$.

El **m promedio** $\langle m \rangle$ es la suma de los m para cada cubeta dividido el número de cubetas ocupadas, donde m es el número de cubetas ocupadas a recorrer para insertar el elemento. Si hay secuencias de cubetas ocupadas contiguas, α puede ser distinto de n/B .

Si todas las cubetas tienen la misma probabilidad de estar ocupadas α , entonces la probabilidad de que al insertar el elemento ésta este libre ($m = 0$ intentos) es $P(0) = 1 - \alpha$. Para que tengamos que hacer un solo intento debe ocurrir que la primera esté llena y la siguiente vacía, y la probabilidad de que esto ocurra es $P(1) = \alpha(1 - \alpha)$. Luego, la probabilidad de que haya que hacer m intentos es $P(m) = \alpha^m(1 - \alpha)$.

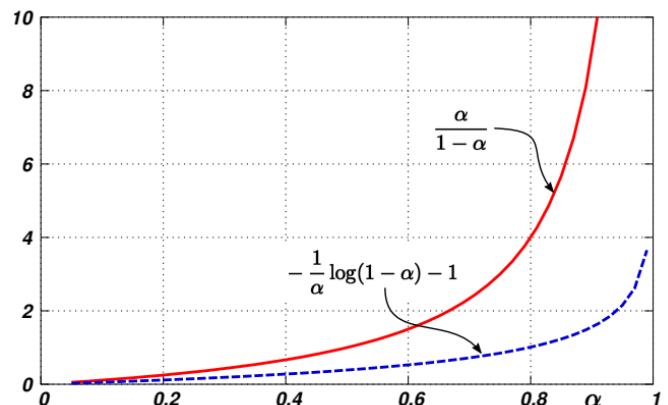
La cantidad de intentos en promedio será entonces $\langle m \rangle = \sum_{m=0}^{B-1} m P(m) = \sum_{m=0}^{B-1} m \alpha^m(1 - \alpha)$. Suponiendo que B es relativamente grande y α no está demasiado cerca de uno, podemos reemplazar la suma por una suma hasta $m = \infty$, ya que los términos decrecen muy fuertemente al crecer m :

$$\langle m \rangle = \sum_{m=0}^{\infty} (1 - \alpha)\alpha \frac{d}{d\alpha} \alpha^m = (1 - \alpha)\alpha \frac{d}{d\alpha} \sum_{m=0}^{\infty} \alpha^m = \frac{\alpha}{1 - \alpha}$$

COSTO DE LA INSERCIÓN NO EXITOSA:

Una **inserción no exitosa** es cuando al insertar un elemento este ya está en el diccionario y por lo tanto en realidad no hay que insertarlo. El costo en este caso es menor ya que no necesariamente hay que llegar hasta una cubeta vacía, el elemento puede estar en cualquiera de las cubetas ocupadas. El número de intentos también depende de en qué momento fue insertado el elemento: si fue en los primeros momentos, cuando la tabla estaba vacía, es menos probable que el elemento esté en el segmento final de largas secuencias de cubetas llenas.

Si la tabla tiene una tasa de ocupación α , entonces un elemento x puede haber sido insertado en cualquier momento previo en el cual la tasa era α' con $0 \leq \alpha' \leq \alpha$. Si fue insertado al principio ($\alpha' \approx 0$) entonces habrá que hacer pocos intentos infructuosos para encontrarlo, si fue insertado recientemente ($\alpha' \approx \alpha$) entonces habrá que hacer el mismo número promedio de intentos infructuosos que el que hay que hacer ahora para insertar un elemento nuevo, es decir $\alpha/(1 - \alpha)$. Si asumimos que el elemento pudo haber sido insertado en cualquier momento cunado $0 \leq \alpha' \leq \alpha$, entonces el número de intentos promedio será $\langle m_{n.e.} \rangle = \frac{1}{\alpha} \int_{\alpha'}^{\alpha} \langle m \rangle d\alpha'$ y reemplazando $\langle m \rangle$ tenemos que $\langle m_{n.e.} \rangle = -\frac{1}{\alpha} \log(1 - \alpha) - 1$.



Eficiencia de las tablas de dispersión cerradas.
 Muestra el número de intentos promedio de la inserción exitosa y la inserción no exitosa.

COSTO DE LA BÚSQUEDA:

El costo de $p = \text{find}(k)$ es $O(1 + \langle m \rangle)$, donde $\langle m \rangle$ es el número de intentos infructuosos. Si el elemento no está en el diccionario (**búsqueda no exitosa**), entonces el número de intentos infructuosos es igual al de intentos infructuosos para inserción exitosa. Por otra parte, si el elemento está en el diccionario (**búsqueda exitosa**), el número de intentos infructuosos es sensiblemente menor, ya que, si el elemento fue insertado al comienzo, cuando la tabla estaba vacía es probable que esté en las primeras cubetas, bien cerca de $h(x)$. El análisis es similar al de la inserción no exitosa y resulta ser $\langle m_{busq.n.e.} \rangle = \frac{\alpha}{1-\alpha}$, $\langle m_{busq.e.} \rangle = -\frac{1}{\alpha} \log(1 - \alpha) - 1$.

COSTO DE LA SUPRESIÓN:

Al eliminar un elemento, no debemos reemplazar el elemento por un `undef`, sino introducir otro elemento `deleted` (“eliminado”) que marcará posiciones donde previamente hubo alguna vez un elemento que fue eliminado. Por ejemplo, para enteros positivos podríamos usar `undef = 0` y `deleted = -1`. Ahora, al hacer $p = \text{find}(x)$ debemos recorrer las cubetas siguientes a $h(x)$ hasta encontrar x o un elemento `undef`. Los elementos `deleted` son tratados como una cubeta ocupada más. Sin embargo, al hacer `insert(x)` de un nuevo elemento, podemos insertarlo en posiciones `deleted` además de `undef`.

Ahora la tasa de ocupación debe incluir a los elementos `deleted`, es decir, en base a la “tasa de ocupación efectiva” α' dada por $\alpha' = \frac{n+n_{del}}{B}$.

Si el destino de la tabla es para insertar una cierta cantidad de elementos y luego realizar muchas consultas, pero pocas inserciones/supresiones, entonces el esquema presentado hasta aquí es razonable. Sin embargo, si el ciclo de inserción-supresión sobre la tabla va a ser continuo, el incremento en el número de elementos `deleted` causará tal deterioro en la eficiencia que no permitirá un uso práctico en la tabla. Luego, se busca “**reinsertar**” los elementos, es decir, se extraen todos los elementos guardándolos en un contenedor auxiliar limpiando la tabla y reinsertando todos los elementos del contenedor.

Como inicialmente la nueva tabla tendrá todos los elementos `undef`, y las inserciones no generan elementos `deleted`, la tabla reinsertada estará libre de `deleted`s. Esta tarea es $O(B+n)$ y se ve compensada por el tiempo que se ahorrará en unas pocas operaciones. La tasa de suprimidos $\beta = \frac{n_{del}}{B}$ es útil para controlar cuando se dispara la reinserción. Cuando $\beta \approx 1 - \alpha$, quiere decir que de la fracción de cubetas no ocupadas $1 - \alpha$, una gran cantidad de ellas dada por la fracción β está ocupada por suprimidos, degradando la eficiencia de la tabla.

En la **reinserción continua** o **redispersión continua**, cada vez que hacemos un borrado hacemos una serie de operaciones para dejar la tabla sin ningún `deleted`. Recordemos que al eliminar un elemento no podemos directamente insertar un `undef` en su lugar ya que no se encontrarán los elementos que están después del elemento insertado y hasta el siguiente `undef`. Luego, si queremos eliminar un elemento, eliminamos temporalmente los elementos que le siguen (hasta el siguiente `undef`) guardándolos en una pila auxiliar S y reemplazándolos por un `undef`. Luego, se elimina también el elemento que se quiere eliminar y se reemplaza por `undef`. Finalmente, se reinsertan todos los elementos que están en S , de nuevo en la tabla.

- Si se usa reinserción continua, entonces no hay en ningún momento elementos `deleted` y el número de intentos infructuosos es $\langle m \rangle = \frac{\alpha}{1-\alpha}$. Las operaciones `find(x)` e `insert(x)` son ambas $O(1 + \langle m \rangle)$, pero `erase(x)` es $O(1 + \langle m \rangle^2)$ ya que al reinsertar los elementos hay en promedio $\langle m \rangle$ llamadas al `insert()` el cuál es a su vez $O(1 + \langle m \rangle)$.
- En la reinserción discontinua, el costo de las tres operaciones es $O(1 + \langle m \rangle)$, pero $\langle m \rangle = \frac{\alpha}{1-\alpha}$, es decir que la tasa de ocupación efectiva crece con el número de elementos `deleted`.

ESTRATEGIAS DE REDISPERSIÓN:

La redispersión permite resolver colisiones, buscando en las cubetas $h(x) + j$, para $j = 0, 1, \dots, B - 1$, hasta encontrar una cubeta libre. A esta estrategia se le llama “de redispersión lineal”. Para evitar que se formen secuencias de cubetas ocupadas, se puede tratar de no dispersar las cubetas de forma lineal sino de la forma $h(x) + d_j$, donde d_0 es 0 y $d_j, j = 1, \dots, B - 1$ es una permutación de los números 1 a $B - 1$. Por ejemplo, si $B = 8$, podríamos tomar una permutación aleatoria como $d_j = 7, 2, 5, 4, 1, 0, 3, 6$. Son como números aleatorios pero que se mantienen durante todo el uso de la tabla.

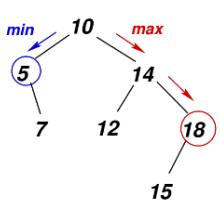
ÁRBOLES BINARIOS DE BÚSQUEDA

Una forma muy eficiente de representar conjuntos son los **Árboles Binarios de Búsqueda** (ABB) (en inglés BST por Binary Search Tree). Un árbol binario es un ABB si:

- Es vacío, o:
- Todos los elementos en los nodos del subárbol izquierdo son menores que el nodo raíz.
- Todos los elementos en los nodos del subárbol derecho son mayores que el nodo raíz.
- Los subárboles del hijo derecho e izquierdo son a su vez ABB.

```
void inorder(btree<int>&T,btree<int>::iterator it,list<int>&L){  
    // Genera el Listado en orden simétrico  
    if(it == T.end())  
        return;  
    inorder(T,it.left(),L);  
    L.push_back(*it);  
    inorder(T,it.right(),L);  
}
```

El **listado simétrico (inorden)** da la lista ordenada de los elementos del árbol. Esto sirve para verificar si un `btree` es ABB, ya que si obtenemos el `inorden` en una lista y la lista no está ordenada de menor a mayor, no es ABB. Que el `inorden` esté ordenado es un sí y sólo si.



FIND, MAX, MIN: Tanto find como encontrar el elemento máximo o mínimo son de orden $O(\log_2 n)$ ya que siguen un camino en el árbol. El mínimo busca siempre por la izquierda hasta llegar al end, el máximo por la derecha hasta llegar al end, y el find compara el nodo con el valor buscado y decide si ir por derecha o izquierda hasta llegar al end.

Un **AB completo** es aquél AB en el cual todos los niveles están completos. Esto es, la cantidad de nodos es $n = 2^l - 1$ donde l es la cantidad de niveles (o sea, si hay niveles 0, 1 y 2, son 3 niveles y $l = 3$). La máxima longitud de caminos es $l = \log_2(n+1) = O(\log_2 n)$.

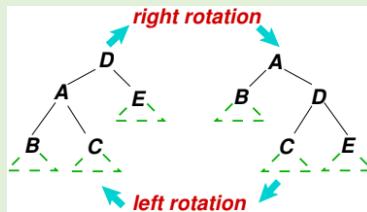
Un **AB bien balanceado** es aquél en el que la longitud media de los caminos es $<l> = O(\log n)$ (si bien no necesariamente es completo).

INSERT Y ERASE: Insertar es $O(\log_2 n)$ ya que sólo implica hacer el find y luego hacer un insert ($O(1)$) en la posición obtenida. Borrar un elemento depende del número de hijos y si el elemento está o no está: si no está o no tiene hijos, $O(\log_2 n)$ ya que o no tengo que hacer nada o tengo que hacer un erase que es de $O(1)$. Si quiero borrar un elemento con un solo hijo, lo borro y subo el subárbol del hijo a la posición del elemento. Si el elemento tiene dos hijos, busco el mínimo del subárbol del hijo derecho o el máximo de la rama izquierda.

RECORRIDO EN EL ÁRBOL: begin() está en la raíz del árbol. Luego, si hago $n++$ se procede según la cantidad de hijos del nodo. Si n tiene hijo derecho, el siguiente es el mínimo de la rama derecha. Si el nodo no tiene hijo derecho, entonces busca en el camino que llega al nodo el último "padre derecho" (es decir, que el nodo siguiente en el camino es hijo izquierdo). El padre derecho puede ser hijo izquierdo o hijo derecho. Si bajé siempre por derecha y el nodo no tiene hijo derecho, es el máximo del conjunto y el next es end().

BALANCEO DEL ABB: La eficiencia del ABB está dada directamente por el "balanceo" del mismo, es decir, por cuanto se desvía la altura del árbol, con respecto a la media $\propto \log n$.

La estrategia que vamos a ver son los **AVL trees**, que mantienen el árbol balanceado mediante rotaciones al insertar/suprimir. Se garantiza que la altura del subárbol izquierdo de un nodo difiere a lo sumo en +1, -1 con respecto a la del derecho.



Método	$T(n)$ (promedio)	$T(n)$ (peor caso)
*p, end()	$O(1)$	$O(1)$
insert(x), find(x), erase(p), erase(x), begin(), p++, ++p	$O(\log n)$	$O(n)$
set_union(A,B,C), set_intersection(A,B,C), set_difference(A,B,C)	$O(n \log n)$	$O(n^2)$

ORDENAMIENTO

El proceso de ordenar elementos ("sorting") en base a alguna relación de orden. Nos concentraremos en el **ordenamiento interno**, es decir, cuando todo el contenedor reside en la memoria principal de la computadora. El **ordenamiento externo** es cuando el volumen de datos a ordenar es de tal magnitud que requiere el uso de memoria auxiliar.

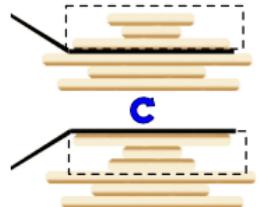
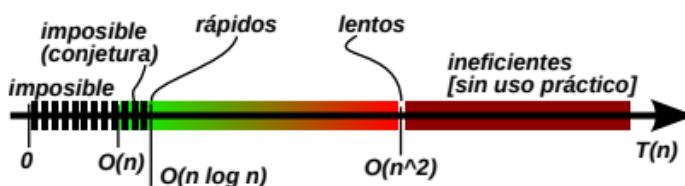
Asumiremos normalmente que los elementos a ordenar pertenecen a algún tipo `key_t` con una **relación de orden** $<$ y que están almacenados en un contenedor lineal (vector o lista). Recordar que $\text{vector}<>$ es de acceso aleatorio y acceder al elemento j -ésimo es $O(1)$, mientras que para lista esto es $O(j)$. Por otra parte, para listas es muy simple dividirlas y unirlas dinámicamente, luego hay técnicas para ambos contenedores.

El **objetivo principal** es obtener algoritmos de ordenamiento lo más eficientes posibles, en **tiempo de cálculo y memoria requerida**. Los tiempos de ejecución se compararán a su crecimiento con el tamaño del contenedor n . Si el ordenamiento se hace sin requerir ningún tipo de memoria adicional (que crezca con n), decimos que es "in-place". Si no, se debe tener en cuenta que también como crece la cantidad de memoria requerida.

Como piso tenemos $T(n) \geq O(n)$, ya que al menos hay que revisar los elementos a ordenar. No puede haber un algoritmo de ordenamiento con tiempo de ejecución menor que $O(n)$, sería falso. Por ejemplo, el método de la burbuja es $O(n^2)$ que se determina un algoritmo lento, pero existen algoritmos más rápidos y otros intermedios. Se conjectura que no hay algoritmos generales de ordenamiento con $T(n) < O(n \log n)$.

Una **relación de orden fuerte** debe ser transitiva: $a < b$ y $b < c$ implica $a < c$ y satisfacer que, dados x e y sólo una de las siguientes es verdadera: $x < y$, $y < x$, $x = y$.

Una **relación de orden** es **débil** si para todo x , y nunca ocurre que $x < y$, e $y < x$ son verdaderas al mismo tiempo. Si ninguna de $x < y$, $y < x$ son verdaderas entonces decimos que x e y son equivalentes.



ALGORITMOS DE ORDENAMIENTO INEFICIENTES:

- **Bogosort:** Desordenar aleatoriamente el vector hasta que quede ordenado. En promedio, $T(n) = O(n \cdot n!)$, peor caso: $T(n) = \infty$.
- **Pancake sort:** Se restringe a hacer solo operaciones en las cuales un rango del vector prefijo ($[0, m]$ con $m \leq n$) es invertido. Sólo tiene interés si se puede encontrar un hardware que realice las inversiones en forma muy eficiente.