Algoritmos y Estructuras de Datos II

Tomás Agustín Hernández

1. Especificación

Consideraciones importantes / Reminders

- Utilizar operadores luego: Si estoy en LPO (Lógica de Primer Orden) utilizar los operadores luego si vemos que hay una posible indefinición como una división, o ingresar a una lista a un índice. Recordar que el para todo y un existe, aunque esté acotado por un rango, los cuantificadores predican IGUAL para todos los valores. Entonces, aunque diga que x es positivo, también probará dividir inclusive por 0 y estallará.
- Recordar las condiciones bidireccionales
  - Si por algún motivo tengo que armar una "lista", como, por ejemplo, los divisores de un número x tengo que indicar que, si el número divide a x, entonces ese número está en res, pero además todos los valores que están en res DIVIDEN a x. Es una condición bidireccional.
  - Otro ejemplo puede ser que tenga que considerar el máximo de una lista, si todos los valores y que están en la lista son menores que res entonces significa que res también pertenece a esa lista original.
- Recordar el significado de los cuantificadores con dos variables al mismo tiempo: En la lógica se ejecutan todos de uno a la vez. Es decir, si tengo que poner un para todo adentro de un para todo entonces hago un para todo solo con dos variables y listo.
- Recordar que cuando en un procedimiento llamo a un predicado y ese predicado devuelve algo de un para todo, existe (básicamente un valor de verdad) tengo que castear ese valor en el procedimiento porque son dos mundos distintos. Ej: asegura: res = True $\iff$ predicado
- Los predicados y funciones auxiliares no describen problemas. Son herramientas sintácticas para descomponer predicados.
  - Los procedimientos pueden llamar a funciones auxiliares o predicados. Un procedimiento no puede llamar a otro procedimiento.
  - Los predicados pueden llamar a predicados o auxiliares.
  - Las auxiliares solo pueden llamar auxiliares.
- No usamos nunca == en especificación, usamos siempre = y estamos comparando, no asignando.
- No existe el guardar o asignar en el mundo de la lógica. No puedo guardar en una lista en un índice específico porque si un valor. Para esto solemos usar que x valor pertenecerá a esta lista, por ejemplo.
- Si tengo un algoritmo que cumple una funcionalidad específica con un require más débil, puedo poner el require más restrictivo y va a funcionar igual pero NO al revés.

Fórmulas compuestas

Decimos que una fórmula es compuesta a una fórmula que tiene más de una operación y esa operación necesita realizarse antes de conocer su valor.

- $(p \land q) \lor m$
- $((p \land q) \lor m) \implies n$

Fórmula atómica

Decimos que una fórmula es atómica si se puede inferir su valor con una, o ninguna operación. Es irreducible.

- p
- $p \land q$

Fórmulas bien definidas

Decimos que una fórmula está bien definida cuando el orden que hay que hacer las operaciones es clara. Es decir, cuando cada operación toma dos variables proposicionales, y al realizar la operación termina siendo una fórmula atómica.

- $p \land q \lor r$ está mal formada. No se especifica si primero se realiza el $\land$ o el $\lor$.
- $(p \land q) \lor r$ está bien formada.
- $p \land q \land r \land m$ está bien formada porque son todas conjunciones.
- $p \lor q \lor r \lor m$ está bien formada porque son todas disyunciones.

Cuantificadores

- Para todo: $\forall$
  - *Garantiza la* conjunción : $p(1) \land p(2) \land p(3) \cdots \land p(m)$. Todos los casos deben ser true para que el cuantificador sea true.
  - Se acompaña por un $\longrightarrow$ a la hora de predicar sobre los elementos.
  - $(\forall i : \mathbb{Z})(0 \leq i < |s| \longrightarrow_L s[i] \ mod \ 2 = 0)$. Todos los elementos de la lista son divisibles por 2.
  - Estructura: $\forall$ + rango + $\longrightarrow_L$
- Existe: $\exists$
  - *Garantiza la* disyunción : $p(1) \lor p(2) \lor p(3) \cdots \lor p(m)$. Con un caso true el cuantificador es true.
  - Se acompaña por un $\land$ a la hora de predicar sobre los elementos.
  - $(\exists i : \mathbb{Z})(0 \leq i < |s| \land_L s[i] \geq 0)$. Existe algún elemento en la lista que es mayor o igual a 0.
  - $\exists$ + rango + $\land_L$

Equivalencias entre fórmulas

Decimos que dos fórmulas son equivalentes $\iff$ los valores de la tabla de verdad al aplicar la operación arroja el mismo resultado.

Valuaciones

Las valuaciones surgen en base a la tabla de verdad. Las valuaciones serian darle valor a las variables proposicionales y ver el resultado de la operación. Solo hacen referencias a fórmulas atómicas.

Tautologias, contradicciones y contingencias

- Una fórmula es tautología $\iff$ el resultado de la operación en cada fila arroja siempre V.
- Una fórmula es contradicción $\iff$ el resultado de la operación en cada fila arroja siempre F.
- Una fórmula es contradicción $\iff$ el resultado de la operación en cada fila arroja siempre V y F.

Relaciones de fuerza entre fórmulas

Decimos que una fórmula es más fuerte que la otra $\iff$ una fórmula es más restrictiva que la otra, o está incluida en la otra.
En el mundo de la lógica, decimos que A es más fuerte que B $\iff A \implies B$

- Si $(A \implies B)$ y $(B \implies A)$ son tautologías, entonces A y B son equivalentes.
- Si $(A \implies B)$ es tautología y $(B \not\Longrightarrow A)$ no es tautología, entonces decimos que A es más fuerte que B.
- Si $(A \not\Longrightarrow B)$ y $(B \not\Longrightarrow A)$ son contigencias, entonces no existe relación de fuerza entre A y B.

Algunos ejemplos:

- $|s| = 0 \implies |s| \geq 0$. En este caso vemos que $|s| = 0$ es más fuerte que $|s| \geq 0$ pues $|s| = 0$ está incluido en $|s| \geq 0$. Por lo tanto, $A \implies B$
- $|s| = 0 \implies |s| \geq 3$. En este caso vemos que $|s| = 0$ no es más fuerte que $|s| \geq 3$ pues $|s| = 0$ no está incluido en $|s| \geq 3$. Por lo tanto, $A \not\Longrightarrow B$
- $2 \leq i < |s| \implies 1 \leq i < |s|$. En este caso A $\implies$ B, pues i = 2 está incluido en el rango de B. Por lo tanto, $A \implies B$
- $0 \leq i < |s| \implies 1 \leq i < |s|$. En este caso A $\not\Longrightarrow$ B, pues el 0 de A no es parte de B. Por lo tanto, $A \not\Longrightarrow B$

Tipos de parámetros en especificacion

- in: Solo nos interesa el valor de entrada de una variable. No la vamos a modificar. Ya están inicializados
- out: Donde se retornará el resultado. No nos importa el valor inicial ni tampoco determina nada en nuestra función.
- inout: Necesitamos el valor original aunque lo terminamos modificando y devolviendo.

Lógica trivaluada

También llamada lógica secuencial porque se procesa de izquierda a derecha; Nos introduce los conceptos de $\wedge_L$ $\vee_L$ $\longrightarrow_L$ y el valor de indefinido $\perp$.

Se termina de evaluar una expresión cuando se puede deducir el valor de verdad.

Considere $x = \text{true} \wedge y = \perp \wedge z = \text{false}$

- $x \vee_L y$: Como el $\vee_L$ necesita uno solo para ser verdadero, entonces como x ya es true entonces toda la fórmula es verdadera.
- $x \wedge_L y$: Como el $\wedge_L$ necesita que ambas variables sean verdaderas, evalúa indefinido y el programa estalla.
- $\neg x \longrightarrow_L y$: Como el $\longrightarrow_L$ solo es falso si el antecedente es true y el consecuente false, como en este caso el antecedente ya es falso, toda la implicación es verdadera.
- $(x \wedge z) \wedge_L y$: Como el $\wedge_L$ necesita que ambas fórmulas sean true, en este caso, como $(x \wedge z)$ es falso, entonces ya toda la fórmula es falsa. Nótese que el $\wedge$ de la condición interna no contiene el luego porque jamás se indefinirá.
- $(\forall i : \mathbb{Z})(0 \leq i < |s| \longrightarrow_L s[i] \geq 0)$ Nótese que aquí usamos un $\longrightarrow_L$ porque podría ser que la lista esté indefinida o no exista el valor en s[i]

Predicados

- Viven en el mundo de la lógica.
- Nos sirven para poder modularizar nuestras especificaciones.
- Solamente devuelven valores de verdad True y False y es necesario castearlos en caso de querer devolver bool como tipo de dato.
- Los predicados pueden llamar a otros predicados o funciones auxiliares.
- Pueden utilizar cuantificadores.
- No tienen requiere ni asegura.
- No admite parámetros in, out, inout.

Ejemplo cuando tenemos que transformar el valor de verdad a tipo de dato:

```
pred divisiblePorDos (n: ℤ) {
    n mod 2 = 0
}
proc esMultiploDeDos (in n: ℤ) : Bool
      requiere {true}
      asegura {res = true ⟺ divisiblePorDos(n)}
```

Ejemplo usando un predicado sin necesidad de transformar el valor de verdad a tipo de dato:

```
   pred todosSonPares (l: seq⟨ℤ⟩) {
    (∀i : ℤ) (0 ≤ i < |l| ⟶_L l[i]mod2 = 0)
}
proc todosPares (in l: seq⟨ℤ⟩) : Bool
      requiere {todosSonPares(l)}
```

Funciones Auxiliares

- Son reemplazos sintácticos.
- Nos ayudan a modularizar las especificaciones.
- No pueden ser recursivas.
- Solo hacen cuentas.
- No pueden utilizar cuantificadores.
- Pueden llamar a predicados.
- Devuelven un tipo de dato.
- No tienen requiere ni asegura.
- No admite parámetros in, out, inout.

`aux sumar` (n: $\mathbb{Z}$, m: $\mathbb{Z}$) : $\mathbb{Z}$ $= n + m$;
`aux sumarTodos` (s: $seq\langle\mathbb{Z}\rangle$) : $\mathbb{Z}$ $= \sum_{i=0}^{|s|-1} s[i]$;

Aridad

Decimos que una función es de aridad n cuando la función recibe n cantidad de parámetros.

Variables Ligadas y Libres

Las variables son ligadas $\iff$ están dentro de un cuantificador mientras que son libres cuando no lo están.

- $(\forall i : \mathbb{Z})(0 \leq i < |s| \longrightarrow_L n \geq s[i])$ i es una variable ligada mientras que n y s son variables libres.
- $(\exists j : \mathbb{Z})(0 \leq j < |s| \wedge_L n \geq s[i])$ j es una variable ligada mientras que n y s son variables libres.
- $(\forall i : \mathbb{Z})(0 \leq i < |s| \longrightarrow_L n \geq s[i]) \wedge P(i)$ Ojo acá. i es una variable ligada, pero la i que está fuera del cuantificador $P(i)$ no está ligada. Esta última debería ser renombrada para no tener problemas y confusiones.

Cuando tenemos variables ligadas **no** podemos hacer nada sobre ellas, entre esas cosas, no podemos reemplazarlas porque no dependen de nosotros sino de los cuantificadores.

Cuantificadores anidados

Anidamos cuantificadores cuando el rango de las variables es exactamente el mismo.

- $(\forall i, j : \mathbb{Z})(0 \leq i, j < |s| \longrightarrow_L n \geq s[i][j]) \equiv (\forall i : \mathbb{Z})((0 \leq i < |s| \longrightarrow_L (\forall j : \mathbb{Z})(0 \leq j < |s| \longrightarrow_L n \geq s[i][j])))$

Estado

Llamamos estado a los valores de las variables en un punto de ejecución específico. El estado de un programa es importante porque muta al asignar valores a las variables. Cuando necesitamos hablar del estado de una variable en un instante específico, hablamos de **metavariables**

Metavariables

Llamamos metavariable a una variable en un instante dado. Es útil cuando tenemos que predicar como cambio el valor de una variable con respecto al inicial.
Cuando tenemos que utilizar metavariables, sea S una variable cualquiera podemos referirnos al instante de tiempo de S como S_t donde t indica el momento.

Notación $S = S_0$

`proc multiplicarPorDosAImpares` (inout l: $seq\langle\mathbb{Z}\rangle$)
`requiere` $\{l = l_0\}$
`asegura` $\{|l| = |l_0|\}$
`asegura` $\{(\forall i : \mathbb{Z})(0 \leq i < |s| \longrightarrow_L if(s_0[i] \ mod \ 2 \neq 0) \ then \ (s[i] = s_0[i] * 2) \ else \ (s[i] = s_0[i]) \ fi)\}$

Nota: Cuando utilizamos metavariables tenemos que indicar que al modificar algo directamente, si no modificamos todo el

conjunto de valores tenemos que indicar que los demás permanecen inalterados. En este caso, como estamos editando los valores, no tendría sentido que la lista salga con mayor longitud, es por eso que garantizamos que no cambia.

Otra manera de resolver el ejemplo anterior es utilizando old(s)

`proc multiplicarPorDosAImpares` (inout l: $seq\langle\mathbb{Z}\rangle$)
`asegura` $\{|l| = |old(l)|\}$
`asegura` $\{(\forall i : \mathbb{Z})(0 \leq i < |s| \longrightarrow_L if(old(s)[i] \ mod \ 2 \neq 0) \ then \ (s[i] = old(s)[i] * 2) \ else \ (s[i] = old(s)[i]) \ fi)\}$

Correctitud de un Programa

Decimos que un programa S es correcto respecto a una especificación si se cumple la precondición P, el programa termina su ejecución y se cumple la postcondición Q.

Tripla de Hoare

Notación para indicar que S es correcto respecto a la especificación (P, Q)

$$\{P\} \ S \ \{Q\}$$

SmallLang

Es un lenguaje que nos permitirá poder validar la correctitud de un programa. Solo tiene dos operaciones:

- x := E ≡ asignación
- skip ≡ no hace nada

Nota: E es una expresión cualquiera. Un valor, una función, cualquier cosa.

Estructuras de Control en SmallLang

- Secuencia de pasos: S1; S2 es un programa $\iff$ S1 y S2 son dos programas.
- Condicionales: if B then S1 else S2 endif es un programa $\iff$ B es una condición lógica (guarda) y S1 y S2 son programas.
- Ciclo: while B do S endwhile es un programa $\iff$ B es una condición lógica y S un programa.

Validez de una tripla de Hoare

$\{x \geq 4\} \ x := x + 1 \ \{x \geq 7\}$ Donde,

- P = $\{x \geq 4\}$
- S = $x := x + 1$
- Q = $\{x \geq 7\}$

¿Vale que $\{P\} \ S \ \{Q\}$? Solo vale $\iff$ $x \geq 6$ por lo tanto, como la precondición P falla en los casos de x = 4, x = 5 podemos decir que la tripla de Hoare no es válida.

Esto que acabamos de hacer se llama demostrar la correctitud de un programa, y acabamos de demostrar que la precondición P para el programa S es demasiado débil pues no nos garantiza que llegaremos a Q cumpliendo P.

Existe una manera formal que nos permite conocer la precondición más débil de un algoritmo.

Predicado def(E)

Dada una expresión E, llamamos def(E) a las condiciones para que E esté definida. Todas las constantes están definidas, por lo tanto def(x) ≡ True. La idea es ir separando en términos e ir colocando las definiciones necesarias para esa operación específica.

- $def(x+1) \equiv def(x) \wedge def(1) \equiv True \wedge True \equiv True$
- $def(x/y) \equiv def(x) \wedge (def(y) \wedge y \neq 0) \equiv True \wedge (True \wedge y > 0) \equiv y \neq 0$
- $def(\sqrt{x}) \equiv (def(x) \wedge x \geq 0)$
- $def(a[i]+3) \equiv (def(a) \wedge def(i)) \wedge_L 0 \leq i < |a| \wedge def(3) \equiv (True \wedge True) \wedge_L 0 \leq i < |a| \wedge True \equiv 0 \leq i < |a|$

Predicado Q_E^x

Cuando hablamos de este predicado hablamos de reemplazar las ocurrencias de x por E en el programa. Solo se reemplazan las ocurrencias libres, no las ligadas.

Axiomas

- Axioma 1: $wp(x := E, Q) \equiv def(E) \wedge_L Q_E^x$
- Axioma 2: $wp(skip, Q) \equiv Q$
- Axioma 3: $wp(S1; S2, Q) \equiv wp(S_1, wp(S_2, Q))$
- Axioma 4: $wp(S, Q) \equiv def(B) \wedge_L ((B \wedge wp(S_1, Q)) \vee (\neg B \wedge wp(S_2, Q)))$

Axioma 1 con secuencias

El axioma 1 nos sirve para asignar una expresión a una variable; Sin embargo, si tenemos que guardar algo en una secuencia debemos utilizar el setAt.

- $wp(b[i] := E, Q)$
  $\equiv def(setAt(b, i, E)) \wedge_L Q^{b[i]}_{setAt(b,i,E)}$
  $\equiv (def(b) \wedge def(i) \wedge def(E)) \wedge_L 0 \leq i < |b| \wedge_L Q^{b[i]}_{setAt(b,i,E)}$
  $\equiv 0 \leq i < |b| \wedge_L Q^{b[i]}_{setAt(b,i,E)}$
  $\equiv setAt(b, i, E)[j] = \{E \ si \ i = j, b[j] \ si \ i \neq j\}$

Algunos ejemplos:
$wp(s[i] := s[i-1], Q)$
$wp(setAt(s, i, s[i-1]), Q) \equiv$
$def(setAt(s, i, s[i-1])) \equiv (def(s) \wedge def(i)) \wedge_L 0 \leq i < |s| \wedge_L def(s[i-1]) \equiv$
$0 \leq i < |s| \wedge_L (def(s) \wedge def(i)) \wedge_L 0 \leq i - 1 < |s| \equiv 0 \leq i < |s| \wedge_L 1 \leq i < |s| + 1 \equiv 1 \leq i < |s|$

TODO: Luego mostrar un ejercicio y aclarar que por cada condición se separan n cuantificadores.

Precondición más débil (Weakest Precondition)

Es la precondición más débil que se necesita para poder ejecutar un algoritmo y satisfacer la postcondición Q.

Notación: $wp(S, Q)$ donde S es el programa y Q la postcondición.

Teorema: Una tripla de Hoare $\{P\}\ S\ \{Q\}$ es válida $\iff P \longrightarrow_L wp(S, Q)$.

Sea el siguiente enunciado, calcule la precondición más debil.

- P = $\{x \geq 4\}$
- S = $x := x + 1$
- Q = $\{x \geq 5\}$

$P \longrightarrow_L wp(S, Q) \equiv wp(x := x + 1, x \geq 5) \equiv def(x+1) \wedge_L Q^x_{x+1} \equiv def(x) \wedge_L def(1) \wedge_L x + 1 \geq 5$
$\equiv True \wedge_L True \wedge_L x \geq 4 \equiv x \geq 4$

Luego, $\{x \geq 4\} \longrightarrow_L \{x \geq 4\}$ es true
Por lo tanto, $wp(x := x + 1, x \geq 5) \equiv \{x \geq 4\}$
Finalmente, probamos que para poder satisfacer Q la precondición más debil que cumple P es cuaqluier $x \geq 4$.

Nota importante: Muchas veces puede ser que se nos solicite indicar que *wp* es incorrecta. Cuando se dice esto, significa que son precondiciones válidas pero hay algunas que no son la más débil.

Precondición más débil en ciclos

Consideremos el siguiente ejemplo $\{???\}S\{x = 0\}$ donde S es el siguiente programa

```
1  while(x>0) do
2      x := x–1
3  endwhile
```

Recordemos que esto es una tripla de Hoare, pero no podemos utilizar wp(S, Q) porque el programa es en base a ciclos. La precondición P más débil acá sería que x≥0 porque para cumplir la postcondición Q me da igual si entra al ciclo o no. Eso tiene que quedar siempre claro, cuando estamos hablando en ciclos, si no entra al ciclo y satisface igual ya está.

Predicado H_k

Definimos el predicado H_k para poder controlar la cantidad de iteraciones que hace un ciclo y poder calcular la precondición más débil.

- $H_0(Q) \equiv def(B) \land \neg B \land Q$
- $H_{k+1}(Q) \equiv def(B) \land B \land wp(S, H_k(Q))\ para\ k \geq 0$

Axiomas

Definimos el Axioma 5 para poder verificar la correctitud de ciclos que hacen una cantidad de iteraciones fijas como

$$wp(while\ B\ do\ S\ endwhile, Q) \equiv (\exists_{i \geq 0})(H_i(Q))$$

¿Cual es el problema del Axioma 5 y el Predicado H_k? El problema es que ambos nos sirven para poder validar la precondición de un ciclo que sabemos que finaliza luego de n iteraciones.

Predicado I - Invariante

Definimos el Invariante de un ciclo como un predicado que:

- Demuestra que el ciclo realmente funciona y nos ayuda a demostrar la correctitud parcial "si termina el ciclo... entonces se cumple Q"
- Vale antes de entrar al ciclo y luego de salir del ciclo.
- En cada iteración, el invariante debe volver a ser válido. En el medio de la iteración puede que deje de valer.
- Un buen invariante incluye el rango de la(s) variable(s) de control del ciclo.
- Un buen invariante tiene alguna afirmación sobre el acumulador del ciclo.

Teorema del Invariante

Para poder probar que un ciclo es correcto, usamos el teorema del invariante.

- $P \equiv$ Las condiciones del requiere
- $P_c \equiv$ Las precondiciones que necesita el ciclo para poder ingresar, muchas veces son las líneas que están por encima del while y el requiere (¡no todas las del requiere!)
- $I \equiv$ Las condiciones que suceden antes y después de entrar al ciclo
- $B \equiv$ La guarda del ciclo
- $Q_c \equiv$ La postcondición del ciclo

Luego,

- $\{P\}\ S\ \{P_c\}$
- $P_c \implies I$
- $\{I \land B\}\ S\ \{I\}$
- $\{I \land \neg B\} \implies Q_c$

Recordatorio: Si aparece la S es que tenemos que calcular la wp porque es parte de la Tripla de Hoare.
Recordatorio importante: Para todo lo que es wp usamos SmallLang, es decir, cuando tenemos que validar la tripla de Hoare. En todos los demás casos usamos lógica.
Ej: No sería válido tener un if en un invariante.

Teorema de Terminación de Ciclo

Utilizamos el teorema de terminación de ciclo para garantizar que cumpliendo la precondición de un ciclo, el programa siempre termina. Para poder probar esto, necesito una función variante f_v.
Llamamos función variante f_v a una función que es siempre estrictamente decreciente y representa una cantidad que se va reduciendo a lo largo de las iteraciones. La función f_v debe garantizar

- $\{I \land B \land v_0 = f_v\}\ S\ \{f_v < v_0\}$
- $\{I \land f_v \leq 0 \implies \neg B\}$

Reglas generales para validar correctitud de ciclo y terminación

- Cuando tenemos que iterar sobre listas, tendremos un índice i que irá hasta incluida la longitud (pues nos sirve) para negar la guarda, mientras que el j será para usar dentro del ciclo. $(0 \leq i \leq |s| \land_L (0 \leq j < i \longrightarrow_L res = \sum_{j=0}^{i} s[j]))$
- Si tengo algo en I deberá estar en P_c y/o Q_c. Esto es porque cuando tengamos que hacer las implicancias, deberíamos comparar de ambos lados y si el invariante tiene algo que los demás no, es siempre falso.
- Si tengo algo en P_c o Q_c no necesariamente tiene que estar en I
- En Q_c se acostumbra a colocar que $i = |s|$ porque nos ayuda a demostrar la terminación del ciclo.
- En la función variante: i va negada si i crece en el ciclo; i va positivo si i decrece en el ciclo.
- En Q_c rara vez tenemos que hablar de rangos de variables.

TADs (Tipos Abstractos de Datos)

Es un tipo de datos porque define un conjunto de valores y las operaciones que se pueden realizar sobre ellos. Es abstracto ya que para utilizarlos no necesitamos saber como está implementado. Describe el qué y no el cómo. Son una forma de modularizar a nivel de los datos.

Importante: En todo enunciado ambiguo, queda en nosotros interpretarlo y eliminar esas ambiguedades decidiendo como lo vamos a considerar.

Ej: En varios ejercicios te piden que un punto deba cambiarse el centro y hay dos maneras de plantearlo

- Agarrar el centro que vienen por parámetro y sumar coordenada a coordenada con respecto a las de la instancia de mi TAD.
- Agarrar el centro que viene por parámetro y considerarlo como el centro final en la instancia del TAD.

Importante: En un TAD podemos usar todos los mismos tipos de datos de especificación y sus funciones correspondientes.

Observadores

Los observadores son una especie de atributo en un objeto en POO. Nos permiten saber qué valores tienen y en qué momento.

- Sirven para describir el estado de una instancia de un TAD y nos permiten consultar su estado virtual.
- Tenemos que poder observar todas las características que nos interesan de las instancias.
- En un instante de tiempo, el estado de una instancia del TAD estará dado por el estado de todos sus observadores (como un debugger).
- Nos permiten distinguir si dos instancias son distintas.
- Todas las operaciones tienen que poder ser descriptas a partir de los observadores.

Ejemplo de un TAD con observadores de tipo dato:

```
1  TAD lista {
2      obs elems: conj<T>
3      obs cantElems: int
4  }
```

Los observadores también pueden ser funciones auxiliares.

- Son las auxiliares de nuestro lenguaje de especificación
- No pueden tener efectos colaterales ni modificar los parámetros
- Pueden usar tipos de nuestro lenguaje de especificación

Ejemplo de un TAD con observadores de tipo dato:

```
1  TAD lista {
2      obs elems: conj<T>
3      obs estaElem(e: T): bool
4  }
```

Igualdad Observacional

Decimos que dos TADs son iguales $\iff$ todos sus observadores son iguales.
Muchas veces no nos basta con la igualdad por defecto con los observadores, y tenemos que declarar nuestra propia igualdad observacional.

Operaciones de un TAD

Las operaciones de un TAD deben estar especificadas y nos indican qué se puede hacer con una instancia. Antes y una vez aplicada una operación tenemos que hablar del estado en el que quedó el TAD con respecto a sus observadores.

Implementación de un TAD

Módulos

- Son la representación / implementación física en código de un TAD.
- Los módulos implementan el TAD.
- Un procedimiento de un módulo puede llamar a otros procedimientos del módulo.

Variables de Estado

Lo que en los TADs eran observadores, acá son variables de estado. Serán manipuladas por las operaciones mediante el código de los algoritmos. Tipos válidos para variables de estado:

- int, real, bool, char, string
- tupla<T1, T2, T3>, struct<campo1: val1, campo2: val2>
- Array<T> (arrays de tamaño fijo)
- No es posible usar tipos de especificación como conj<T> o seq<T>

El módulo puede tener variables de estado que no hagan referencia a ningún observador de un TAD, por ejemplo: guardar un máximo en un módulo aunque el TAD no lo pida. A veces estas cosas se hacen solo por temas de a la hora de implementarlo en un lenguaje de programación.
Nota: Una variable de estado también puede contener otro módulo.

Invariante de Representación

- Define una restricción sobre el conjunto de valores que pueden tomar las variables de estado para que se considere una instancia válida.
- Es equivalente al invariante en ciclos, pero acá aplicado a TAD's
- Este es inicializado en el constructor una vez realizada la instancia
- Vale siempre antes de llamar a los métodos y luego de cada método. Siempre y cuando tenemos un parámetro inout, tenemos que ver que los cambios que hagamos hagan seguir valiendo a la invariante de representación.
- Todos los métodos pueden asumir que el invariante de representación siempre vale al ser llamados.
- ¡Prestar atencion a las verificaciones bidireccionales! En el sentido de que si tengo $dict < Alarma, Conj < Sensores \gg$ y $dict < Sensor, Conj < Alarmas \gg$ si en $dict < Alarma, ... >$ tengo varios sensores, esos sensores estan en Sensor y ademas, si busco en las alarmas de cada sensor deberia estar la $dict < Alarma, ... >$
- Cuando tengo un diccionario tengo que usar un cuantificador para hablar y tengo que acceder y hablar si o si por la key.

Función de Abstracción

Definimos función de abstracción (α) como la representación física de lo que tiene el código, es decir, cuando acá hablamos de instancia.elems decimos que el código tiene un campo llamado elems. En TAD cuando nosotros hablabamos de un observador podía existir o no en la implementación como atributo pero seguir cumpliendo la especificación.

- Es un predicado
- Toma como parámetro una instancia del módulo y una instancia del TAD.
- Hacemos referencia a observadores y a otros predicados, no procs
- Todos los observadores del TAD deben tener un vinculo con una variable de estado pero no necesariamente al revés
- Para poder escribir, usamos lenguaje de especificación como lo conocemos. Podemos usar por ejemplo —a.data— o a.data.length da igual, pero hay que ser consistentes
- Rara vez se colocan rangos acá, casi nunca, excepto cuando hay alguna variable de estado y observadora que hablan de capacidades.

Es importante considerar que todo lo que está en el invariante de representación ya está implícito en la función de abstracción. Ej: Tenemos un módulo que tiene ventas, mayorProductoVentas, y mayorPrecio pero el TAD solo tiene ventas. En este caso, en el invariante de representación hablamos de mayorProductoVentas y mayorPrecio haciendo las relaciones con ventas (en temas de key del producto) pero en el predicado de astracción NO hablás de mayorProductoVentas y mayorPrecio primero porque no son observadores del TAD pero tampoco hay que garantizar nada porque ya está implicado por el invariante de representación

Operaciones de un Módulo

Como un módulo implementa un TAD, todas sus operaciones deben estar implementados en código. Para escribir las implementaciones de los procs usamos una especie de SmallLang.

- Declaración de variables:
  - var x: int
  - var c: ConjuntoArr¡int¿
- Asignación: x := valor
- Condicional: if condicion then codigo else codigo endif
- Ciclo: while condicion do codigo endwhile
- Llamar a un proc:
  - c.vacio()
  - b : = c.vacio();

Memoria dinámica

Los tipos complejos son usados siempre por referencia. El valor indefinido es identificado por la palabra null. Acceder a algo null explota.
Las variables de tipos complejos deben ser inicializadas mediante el operador new. Tipos nativos:

```
1  var a: Array<int>
2  var b: int
3  if a == null then
4  ...codigo
5  endif
6  a := new Array<int>(10)
7  b := a[0]
```

Tipos de otros módulos:

```
1  var a: ConjArr<int>
2  a := new ConjArr(10) //longitud 10
```

Pasaje por referencia:

```
var a: ConjArr<int>
var b: ConjArr<int>
a := new ConjArr(10); //longitud 10
b := a
```

Nota: El pasaje por referencia, a veces se le conoce comúnmente como aliasing y para evitarlo, hay que crear una nueva instancia con los valores de la otra instancia en vez de asignarla directamente.

Aliasing en procedimientos de Módulo

Siempre hay que recordar que el módulo es lo más cercano a código que tenemos. Por lo tanto, en operaciones como concatenar, copiar, o quizá mover de un lado al otro para luego borrarlo hay que recordar el aliasing.
Porque por ejemplo, si tengo que mover cierta data hacia un lado y luego borrarla de de donde estaba, si la moví por referencia y después la borré, perdí todo.

Contrato entre métodos de un módulo

Los métodos de un Módulo tienen un contrato entre sí. Supongamos que necesitamos que la complejidad de búsqueda de un método de mi módulo sea o(log n) sabiendo que mi lista está ordenada. Aquí, el método busqueda podrá usar la búsqueda binaria para poder hacer el proceso lo más rápido posible, pero ¿qué sucede si al agregar un elemento en el método agregar() la lista de salida que se modifica en el módulo no está ordenada?
El contrato de búsqueda no se cumpliría porque la búsqueda binaria no funcionaría porque necesita que la lista esté ordenada.

En este caso, que la lista esté ordenada debería ser una condición del invariante de representación para evitar estos problemas.

Este módulo es bueno para la búsqueda de un elemento pero el agregar un elemento es mucho más costoso por el tema de reordenar la lista nuevamente.

Mirada al futuro: Los métodos deben cumplir una complejidad algorítimica dada.

Estructuras de Datos

Colección

Representa un grupo de objetos. Provee de una arquitectura para su almacenamiento y manipulación.

- Secuencia
- Conjunto
- Multiconjunto
- Diccionario

¿Por qué una tupla no es parte del grupo? Porque la tupla es fija, una vez definida e inicializada su longitud no cambia.

Tipos paramétricos

Son variables de tipo, es decir, variables con tipo genérico. La manera más común de indicar un genérico son T, K o V. Hay que tener cuidado con las operaciones que se realizan con las variables genéricas porque algunos operadores nos restringuen su uso a ciertos tipos.
Ej: No puedo utilizar X - Y si X, Y son genéricos y nos envían X, Y como char.

Iteradores

Nos permiten recorrer colecciones de una manera abstracta sin saber su estructura.
Un buen iterador se distingue de otro iterador por la velocidad de iterar la estructura.
Operaciones con iteradores:

- ¿Estoy sobre un elemento?
- Obtener el elemento actual
- Avanzar al siguiente elemento
- Retroceder al elemento anterior (sii es bidireccional)

Los Iteradores son una clase privada que van dentro de la clase que queremos que se instancie y se pueda recorrer.
Es privada porque no queremos que sea instanciada más que por la instancia de la clase que queremos recorrer.
Los Iteradores no van hasta n, van hasta n+1.


```
1  public class Vector<T> implements List<T>{
2      private T[] elementos;
3      private int size;
4
5      private class Iterador implements Iterator<T>{
6          int indice;
7
8          Iterador(){
9              indice = 0;
10         }
11         public boolean hasNext(){
12             return indice != size;
13         }
14         public T next(){
15             int i = indice;
16             indice = indice + 1;
17             return elementos[i];
18         }
19     }
20
21     public Iterator<T> iterator(){
22         return new Iterador();
23     }
24 }
25
26 Iterator it = vector.iterator();
27 while(it.hasNext()){
28     System.out.println(it.next());
29 }
```

Singly Linked Lists - Listas simplemente enlazadas

Es una estructura que sirve para representar una secuencia de elementos donde cada elemento es un Nodo que tiene un valor y una referencia al siguiente.
Nota: Si el Nodo actual es el último de la Singly Linked List lo notamos porque el ultimo.siguiente = null. El Nodo debe ser responsable de guardar al siguiente porque caso contrario, no existirá otra referencia al siguiente Nodo. Ventajas de las Singly Linked Lists:

- Mas fino uso de la memoria.
- Insertamos fácilmente al principio y al final.
- El costo de insertar al final es o(n) pues debemos recorrer todos los nodos para insertar uno nuevo. Es por eso que es común guardar el último nodo almacenado en la clase.
- Eficiente para reacomodar elementos.
- Es malo en rendimiento o(n) si necesito buscar un elemento en específico. Es decir, perdemos el acceso aleatorio a los elementos.

Double Linked Lists - Listas doblemente enlazadas

Exactamente igual que la Singly Linked Lists pero acá los Nodos tienen almacenado: Referencia al nodo previo, valor y referencia al nodo siguiente.
Nota: Si el Nodo actual es el último de la Double Linked List lo notamos porque el ultimo.siguiente = null.

```
1  public class DoubleLinkedList<T> {
2      private Nodo primero;
3      private int longitud;
4
5      private class Nodo {
6          Nodo prev;
7          Nodo sig;
8          T valor;
9
10         public Nodo(T val){
11             this.val = val;
12         }
13     }
14
15     public DoubleLinkedList(){
16         this.longitud = 0;
17     }
18 }
```

Listas vs Linked Lists

Ambas son bastante rápidas a la hora de iterar sobre ellas, sin embargo:

- Las listas nos permiten acceder rápidamente a un elemento mientras que las Linked Lists no.
- Las Linked Lists nos permiten agregar elementos rápidamente al inicio (O(1)) y no necesitamos reacomodar nada más que decir que el anterior primero ahora es el segundo mientras que en la lista tenemos que reacomodar todos los elementos como antes pero colocar el nuevo al principio (O(n))

Complejidad Algorítmica

Analisis de la Complejidad de Algoritmos

Nos permite elegir entre distintos algoritmos para resolver el mismo problema o distintas formas de implementar un TAD. Esto es importante porque nos permite optimizar:

- Tiempo de ejecución
- Espacio (memoria)
- Cantidad de procesadores (en caso de algoritmos paralelos)
- Utilización de la red de comunicaciones (para algoritmos paralelos)

El análisis se puede hacer de forma experimental o teórica

Ventajas del enfoque teórico:

- Se hace antes de escribir código
- Vale para todas las instancias del problema (no en un caso específico)
- Es independiente del lenguaje de programación
- Es independiente de la máquina donde se ejecuta (el tiempo no varía)
- Es independiente del programador

Análisis Teórico

- Se realiza en función del tamaño del input
- Para distintos tipos de input
- Análisis asintótico

Operaciones Elementales

T(l) será una función que mida el número de operaciones elementales requeridas para la instancia l. Las operaciones elementales (OE) serán aquellas que el procesador realiza en tiempo acotado por una constante (no depende del tamaño de la entrada). Ej: x = 0, if(x = 2). Es decir, podemos generalizar las operaciones elementales a:

- Operaciones aritméticas básicas (suma, resta, división, multiplicación)
- Comparaciones y/o operaciones lógicas
- Acceder a elemento de array
- Asignaciones a variables de tipos básicos (las inicializaciones no consumen tiempo)

Cálculo de Operaciones Elementales

- T(If C Then S1 Else S2 Endif;) = T(C) + max{t(S1), t(S2)}
- T(Case C Of v1:S1 | v2:S2 | ... | vn:Sn End;) = T(C) + max{t(S1), t(S2), ..., T(Sn)}
- T(While C Do S End;) = T(c) + (nº iteraciones) * (T(S) + T(C))
- T(MiFuncion(P1, P2, ..., Pn)) = 1 + T(P1) + T(P2) + ... + T(Pn)

Tamaño de la entrada

- T(n): complejidad temporal (o en tiempo) para una entrada de tamaño n
- S(n): complejidad espacial para una entrada de tamaño n

Importante: NUNCA hay que restringir la longitud de entrada de una lista; Por ejemplo: no tiene decir que el mejor caso sea que la lista esté vacía, porque la complejidad se da en base a listas de longitud n sin ningún tipo de restricción.

Análisis de los casos

- $T_{mejor}(n) = \min_{instancias\ l,\ |l|\ =\ n}$ {t(l)}. Recorriendo una lista, el mejor caso es que esté en primera posición.
- $T_{peor}(n) = \max_{instancias\ l,\ |l|\ =\ n}$ {t(l)}. Recorriendo una lista, el peor caso es que el elemento no esté.
- $T_{prom}(n)$ = No lo vamos a usar pero, es algo más parecido a estadística.

Cuando el tamaño de datos es grande, los costos de los diferentes algoritmos pueden variar de manera significativa. En tamaño de datos chico, no nos interesa el tiempo de ejecución.

Principio de Invarianza

Si dos algoritmos solo varían en una constante, entonces no nos importa porque la complejidad es la misma.

$$T_1(n) \leq cT_2(n)$$

c: constante real $c > 0$
$n_0 \in N$ tales que $\forall n \geq n_0$

Comportamiento Asintótico

Comportamiento para los valores de la entrada suficientemente grandes.
Nota: Las funciones f y g dependen de un n. No existen funciones constantes menores que O(1).

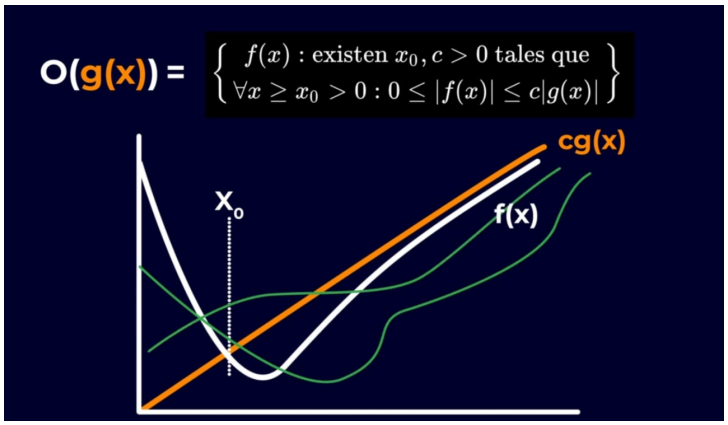
O (cota superior)

Sirve para representar el límite o cota superior del tiempo de ejecución de un algoritmo.
La notación f ∈ O(g) expresa que la función f no crece más rápido que alguna función proporcional a g (g es la cota superior de f).
Ej: $100n^2 + 300n + 1000 \in O(n^2) \wedge 100n^3 + 300n + 1000 \in O(n^2)$

$$f(n) \in O(g(n)) \iff \exists c \in R > 0, n_0 \in N\ tal\ que\ \forall n \geq n_0 : f(n) \leq c * g(n)$$

Las funciones f son aquellas que crecen más lento, ej si me dan O(n) busco funciones que crezcan mas lento que n

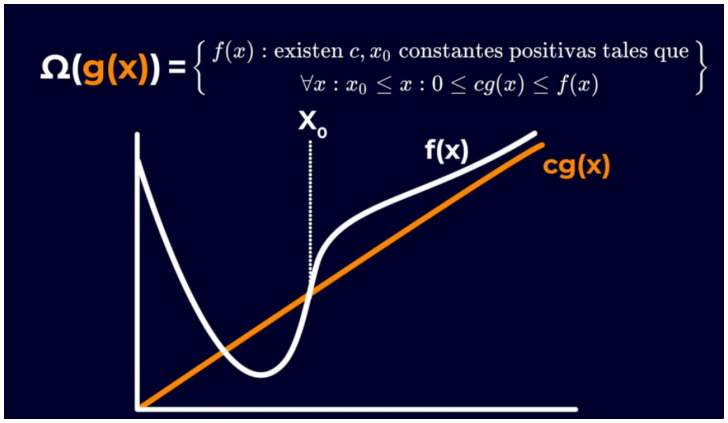
- $n-1 \in O(n)$ porque al ser -1 una constante, no cambia nada.
- $2n \in O(n)$ porque al ser dos veces n (n+n) y es una constante, no cambia nada.
- $n+1 \in O(n)$ porque al ser +1 una constante, no cambia nada.
- $n\ log(n) \in O(n)$
- $log(n) \in O(n)$
- $n^2 \notin O(n)$



Ω (cota inferior)

Sirve para representar el límite o cota inferior del tiempo de ejecución de un algoritmo. La notación f $\in \Omega(g)$ expresa que la función f está acotada inferiormente por alguna función proporcional a g (g es cota inferior de f).
Ej: $100n^2 + 300n + 1000 \in Omega(n^2) \land 100n^2 + 300n + 1000 \in Omega(n)$

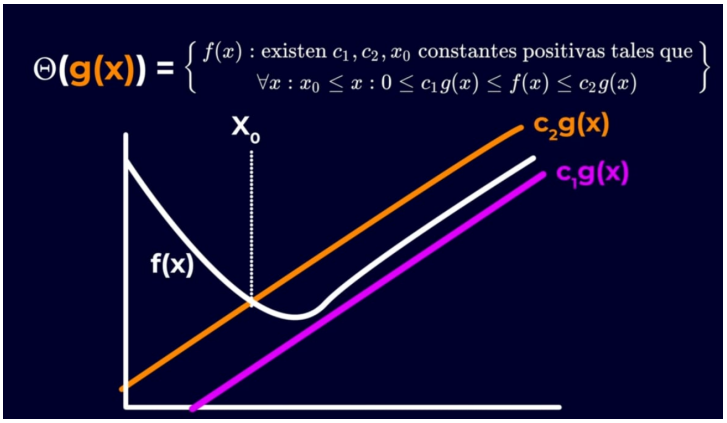
- $n/2 \in \Omega(n)$
- $n-1 \in \Omega(n)$
- $n^2 \in \Omega(n)$
- $n^k \in \Omega(n)$
- $log(n) \notin \Omega(n)$



θ (orden exacto)

$$f(n) \in \theta(g(n)) \iff f(n) \in O(g(n))\ y\ f(n) \in \Omega(g(n)).\ Es\ decir,\ \theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

Básicamente: Tiene que valer en O y en Ω, no debe pasarse de Ω pero tampoco estar por debajo de O.



Definición parecida a inducción corrida en Complejidad

Tanto en la definición de O, Ω y θ se nombra para un $n > n_0$. Esto es porque no todas las funciones cumplirán la definición, sino aquellas que son mayores a n_0. Es por eso que en las imágenes se ven como inicialmente no cumple, pero a partir de un x_0 (n_0) vale para todo n.

Propiedades

- Suma: O(f) + O(g) = O(f+g) = O(maxf,g)
- Producto: O(f) $*$ O(g) = O(f*g)
- Reflexividad: f $\in O(f)$
- Simetría: Solo vale en $\theta = f \in \theta(g) \implies g \in \theta(f)$
- Transitividad: $f \in O(g) \wedge g \in O(h) \implies f \in O(h)$

Intuición de Análisis de Complejidad posibles

- Mejor caso != Peor caso (cuando el ciclo corta antes)
- Mejor caso = Peor caso (cuando el ciclo no tiene condición de corte)
- Cuando tengo ramas if else, la complejidad mayor se dará por la rama que tenga mayor cantidad de operaciones en cuanto a ciclos.
- Cuando tenga guardas como por ejemplo, while $b > 0$ y la variable se va dividiendo entre dos, la complejidad será algo parecida a log n.
- Cuando una función recibe dos parámetros, pero uno de ellos no tiene un rol fundamental en algún ciclo o guarda, entonces raramente esté en el cálculo final de la complejidad.
- Recordar que es normal ver casos donde se aplique la sumatoria de gauss o tengamos que masajear la expresión para que aparezca.
- Si la guarda del ciclo se incrementa de manera: $i * 2$ o $i/2$ entonces la complejidad del ciclo será logarítmica.
- Para poder calcular la complejidad, si tenemos una suma basta con quedarnos con el término más grande. (ej: n + m tiene 2 casos)

Cota Ajustada

Usamos θ para indicar cotas ajustadas al momento de tener que calcular el peor y el mejor caso.

Complejidad Algorítmica en Polinomios

Usamos límites para estos casos

Sean f, g: $\mathbb{N} \implies \mathbb{R}_{>0}$. Si existe:

$$\lim_{n\to\infty} f(x)/g(n) = l \in \mathbb{R}_{\geq 0} \cup \{+\infty\}$$

- $f \in \theta(g) \iff 0 < l < +\infty$
- $f \in O(g)\ y\ f \notin \Omega(g) \iff l = 0$
- $f \in \Omega(g)\ y\ f \notin O(g) \iff l = +\infty$

Complejidad en procedimientos de módulo

Se calculan igual que observando una función dada que hace algo. Lo único que quería aclarar acá, es que siempre para las eliminaciones o búsquedas nunca se usen (en caso de haber) punteros que tengan referencia al último valor. El caso excepcional es el primero, pero el último hay que buscarlo con la forma de recorrer la estructura tradicional que haya.
Véase anexo para ver un ejemplo al respecto.

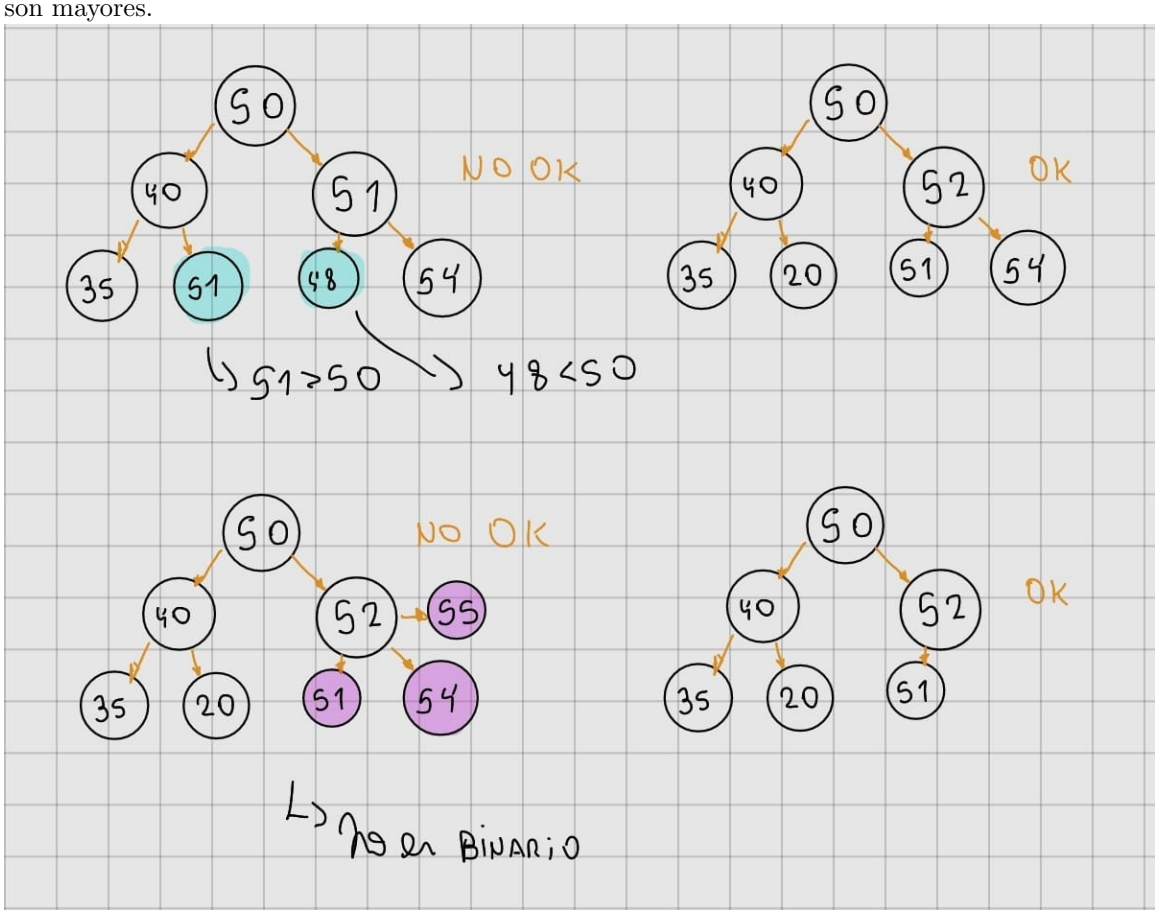
Árboles / Árboles binarios

Se definen recursivamente y sus operaciones necesitan de ella. Utilizamos nodos.
Terminología que vamos a usar: Raíz & Hoja
Para poder obtener un elemento de un árbol se debe empezar por la raíz y recorrer cada una de las hojas recursivamente.

OBS: Las hojas pueden tener o no nodos hijos; si tienen hijos entonces también son árboles.
Árboles binarios Son aquellos en los cuales cada nodo tiene máximo 2 elementos.

Árboles binarios de búsqueda (ABB)

Para todo nodo, los valores del subárbol izquierdo son menores que el valor del nodo y los valores del subárbol derecho son mayores.



Nótese que de ninguna rama derecha puede haber un número mayor a la raíz; de ninguna rama izquierda puede haber un

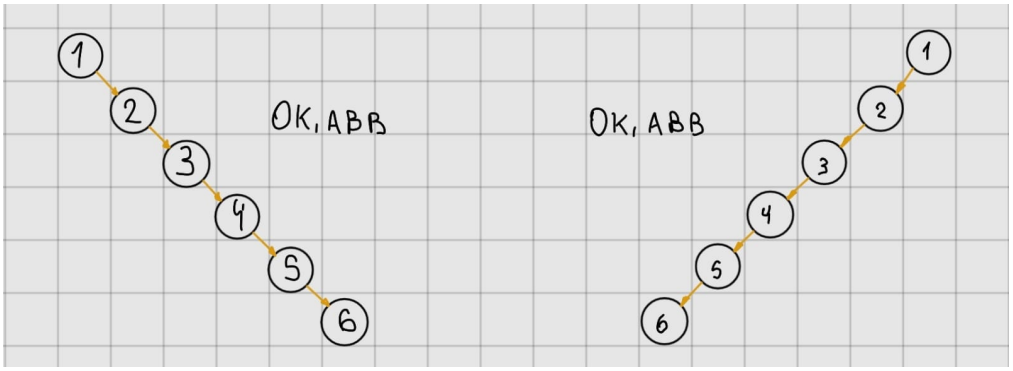
número menor a la raíz.
La complejidad de búsqueda en el peor caso es de $O(logn)$ Algoritmos comúnes en árboles binarios de búsqueda:

- Arbol vacío: Aquel que tiene la raíz en null
- Búsqueda de elemento: Si el nodo es null o el elem es igual al valor del nodo devuelvo n. Caso contrario, si el valor a buscar es menor al dato del nodo entonces busco más a la izquierda, caso contrario busco a la derecha. Cuando busca, habla de llamar a la misma función recursivamente. Complejidad O(n) con n la altura del árbol porque pasa solo una vez por cada nodo.
- Insertar elemento: Primero agarro la raiz del árbol, si la raiz es null entonces no hago nada. Si el árbol no tiene una raíz, entonces mi elem es la raíz; Si ya tiene una raíz me fijo si el elem a insertar es menor o mayor a la raíz, si es mayor entonces voy al primer hijo derecho de la raíz (no entiendo la parte final) dice algo como si el elem¡prev.dato then padre.izq = newnodo osea tiene sentido pero en el pseudocodigo prev no existe enn ningun lado definido. Complejidad O(n) para agregar, aunque si hay distribución uniforme de las claves O(log n)
- Eliminar elemento: hay 3 casos. Sea u una variable cualquiera
  - u es una hoja: Si u es una hoja significa que no tiene hijos, por lo tanto lo puedo eliminar sin necesidad de reordenar nada.
  - u tiene un solo hijo: Si u es una hoja con un solo hijo (a la izq o a la derecha) basta con mover el hijo a la posición del nodo a eliminar.
  - u tiene dos hijos: Si u es un nodo con dos hijos hay que encontrar al predecesor inmediato (v) de u. **v no puede tener dos hijos, en caso contrario no es predecesor inmediato**, una vez encontrado copiar la clave de v en lugar de la de u, borrar el nodo v (acá hay que revisar qué sucede si tiene un hijo, iría al caso 2, caso contrario caso 1). Lo mismo se puede hacer con el sucesor inmediato (mismas reglas que predecesor inmediato).
  - Complejidad O(n) para cualquier tipo de borrado.

Aclaración: Predecesor inmediato (el más grande de los chiquitos), osea rama izquierda busco el último de la der a partir del nodo que estoy.
Aclaración: Sucesor inmediato (el más chiquito de los grandes), osea rama derecha busco el último a la izquierda a partir del nodo que estoy.

ABB con complejidad O(n) en ciertas operaciones

Árbol binario con n nodos anidados todos mayores o todos menores.



¿Cuál es el problema de los ABB? que al hacer inserciones, eliminaciones o ver si un elemento dado pertenece al árbol, el costo en el peor caso es O(n).
Para optimizar esto, existen los AVL que nos ponen restricciones en base a la cantidad de nodos que puede haber anidados en base ciertas reglas de balanceo.

ABB balanceados (AVL)

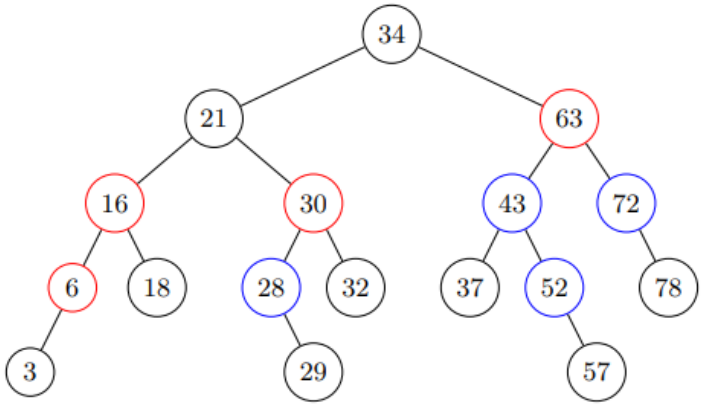
Un árbol AVL es un ABB balanceado en altura. Un árbol binario perfectamente balanceado de n nodos tiene una altura de: $log_2(n) + 1$ donde las hojas son más del 50 % de los nodos
La inserción y el borrado se hace exactamente igual que en un ABB pero acá, como extra se hace el rebalanceo.
Se define el factor de balanceo de un nodo v de un árbol binario como:

$$fb(v) = A_D - A_I$$

donde A_x es la altura del subárbol x
Existen 3 posibles factores de balanceo para cada nodo:

- -1: La cantidad de ramas de la izquierda del nodo son más que la rama derecha.
- 0: La cantidad de ramas a la izquierda o derecha del nodo son la misma cantidad.
- 1: La cantidad de ramas de la derecha del nodo son más que la rama izquierda.

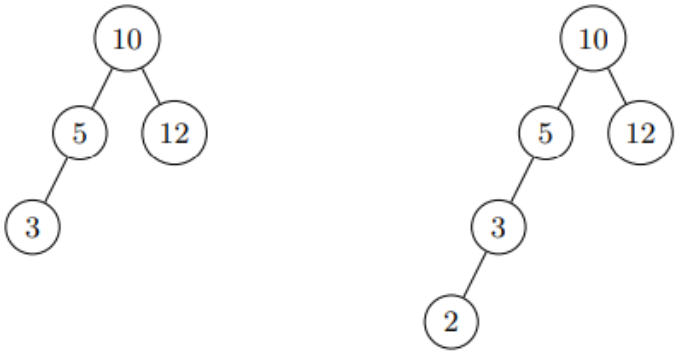
Llamamos entonces, árbol balanceado para cuando para cualquier nodo en él, la diferencia de longitud entre sus ramas izquierda y derecha difiere, a lo sumo, en una unidad.
Nota: los caminos no nos importan mucho, considerar cada nodo como un árbol distinto. Si los tengo en misma altura pero uno está más abajo que otro, está desbalanceado.



- Negro: Factor de balanceo 0.
- Rojo: Factor de balanceo -1.
- Azul: Factor de balanceo 1.

Mantener balanceo de AVL al insertar o eliminar

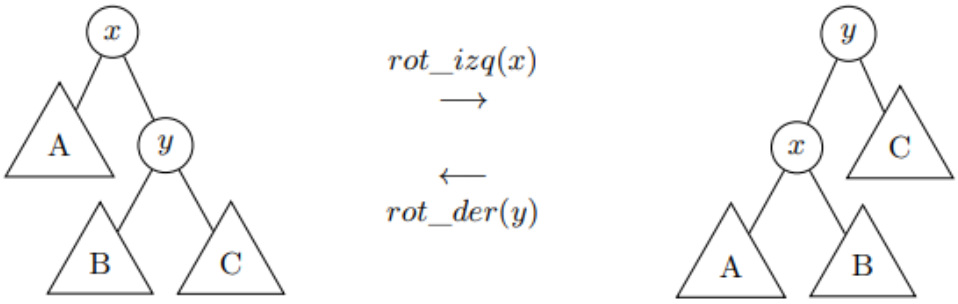
Una de las dificultades más grandes de mantener el AVL invariante luego de cada operación es mantenerlo balanceado. Viendo la figura de la izquierda: es un AVL porque si vemos, la rama de la izquierda del 10 son 2 nodos y la derecha 1 y difieren en un solo nodo, por lo tanto está balanceado.



Viendo la figura de la derecha, no está balanceado: porque la rama de la izquierda del 10 son 3 nodos (altura 3) y la derecha 1 (altura 1), y difieren en 2 nodos, por lo tanto no está balanceado.
¿Cómo solucionamos esto para mantener el invariante? Rotaciones

Rotaciones

Existen muchas rotaciones posibles para un árbol, pero las elegimos dependiendo de la estructura del árbol que tengamos que rebalancear.
Es importante que cada vez que hacemos una rotación, el invariante debe seguir valiendo.



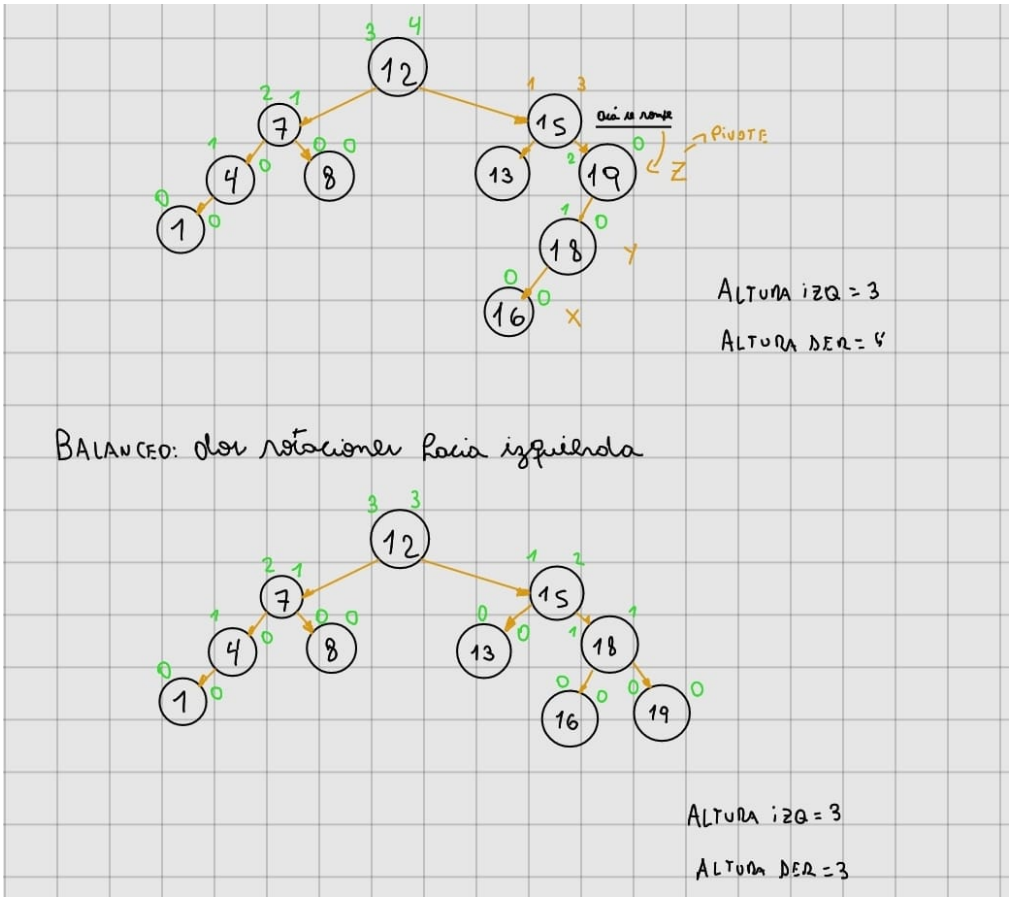
Nótese que al rotar el árbol, sigue valiendo el invariante que $A < x < B < y < C$, esto nos indica que al rotar conseguimos el mismo árbol.

¿Qué rotación tengo que usar?

- Left-Left(LL o II): Ocurre cuando un nodo tiene un subárbol izquierdo que está desbalanceado a la izquierda. La solución es una rotación simple a la derecha. Véase anexo
- Right-Right(RR o DD): Ocurre cuando un nodo tiene un subárbol derecho que está desbalanceado a la derecha. La solución es una rotación simple a la izquierda. Véase anexo
- Left-Right(LR o ID): Ocurre cuando un nodo tiene un subárbol izquierdo que está desbalanceado a la derecha. La solución son dos rotaciones: primero una rotación a la izquierda en el subárbol izquierdo y luego una rotación a la derecha en el nodo. Véase anexo
- Right-Left(RL o DL): Ocurre cuando un nodo tiene un subárbol derecho que está desbalanceado a la izquierda. La solución son dos rotaciones: primero una rotación a la derecha del subárbol derecho y luego una rotación a la izquierda en el nodo. Véase anexo

Consejos útiles:

- Muchas veces, una rotación no siempre es suficiente para rectificar un nodo que está desequilibrado.
- El rebalanceo (realiza las rotaciones) se invoca para todos los nodos de la rama hasta la raíz (hay que mandar el árbol entero).
- El máximo de rotaciones consecutivas para balancear un árbol es de 2.
- Al insertar un nodo en un AVL, el costo es O(log n). El costo de inserción es O(log n) y luego para rebalancear tambien, O(log n).



Cola de Prioridad

Es exactamente igual que el TAD Cola, pero difiere en la forma en que removemos elementos. No quitamos el primero que ingresó, sino que lo sacamos en base a un factor de prioridad que definimos a la hora de armar el TAD.
Si llegase a suceder que un mismo elemento tiene la misma prioridad, debería especificarse cual se quitaría.
La prioridad la expresamos con un entero, pero puede ser cualquier tipo α que pueda ser comparado con un orden $<_\alpha$.

Ej.: Imaginemos que vamos a una guardia, y estamos primeros a punto de ser atendidos pero llega alguien que está en estado grave. Esta persona va a ser atendida antes que nosotros aunque hayamos llegado antes. En ese caso, es una cola de prioridad pues la prioridad está dada por la gravedad del paciente.

Casos de uso:

- Sistemas operativos
- Algoritmos de Scheduling
- Gestión de colas

Véase anexo para un ejercicio de elección de estructuras para armar una cola de prioridad.

Heaps

Es la implementación del TAD ColaPrioridad. Tiene la misma complejidad algorítmica que un árbol AVL pero es más fácil y elegante de implementar.
Invariante de representación:

- Árbol binario perfectamente balanceado (difiere a lo mucho en un único nodo)
- El último nivel está lleno de nodos desde la izquierda (es izquierdista). Cuando el nivel del la rama izquierda ya difiere en uno con la derecha, agrego en las hojas de la rama derecha para dejar el árbol con misma altura, pero la idea es agregar de izquierda a derecha siempre.
- La clave (prioridad) de cada nodo es **mayor o igual** que la de sus hijos (si es que tiene)

- Todo súbarbol es otro heap.
- NO es un ABB. En un ABB el hijo derecho es más grande que el padre. En un Heap ambos hijos de un Nodo son menores.

Min-Heap: Min hace referencia a que el proceso de extraer, se hace sacando el mínimo en O(1). Si sacamos el mínimo en O(1) significa que todos los nodos debajo de la raíz son menores estrictos a él.
Max-Heap: Max hace referencia a que el proceso de extraer, se hace sacando el máximo en O(1). Si sacamos el máximo en O(1) significa que todos los nodos debajo de la raíz son mayores estrictos a él.

Véase operaciones heaps para un ejemplo más visual de las operaciones
Véase implementaciones heaps de maneras diversas.

Complejidad en operaciones con Heaps

Nota: acá hablamos de un max-heap. recordemos que los que están arriba son mayores estrictos que los hijos.

- Próximo: O(1) → raíz del árbol o primer elemento del array.
- Encolar(elemento): O(log n) → Sin necesidad de conocer el elemento, lo que tenemos que tratar es llenar el árbol de manera izquierdista. Hay dos casos
  - Si el elemento que ingresé en el lugar libre es **menor a su padre, dejo todo como está.**
  - Si el elemento que ingresé en el lugar libre es mayor a su padre, **hago el swap con el padre**. Por lo tanto ahora tengo garantizado que el elemento nuevo es el padre y los hijos son menores a este (vuelve a valer el invariante).
  - **Algoritmo**: Insertar elemento al final del heap, y luego subir(elemento)
- Subir(elemento) o Sift-Up: Mientras que el elemento no sea la raíz, y la prioridad del elemento sea mayor al padre, entonces intercambio **el elemento con el padre**.
- Desencolar(elemento): O(log n) → Son varios pasos
  - Intercambio la raíz con el último elemento ingresado.
  - Borro mi anterior raíz.
  - Mi nueva raíz (el último ingresado), lo comparo con sus dos hijos verificando que
    - si estamos en un max-heap, sea el mayor de los dos hijos, si no lo es, entonces intercambio con el MAYOR/IGUAL HIJO. Hago esto recursivamente.
    - si estamos en un min-heap, sea el menor de los dos hijos, si no lo es, entonces intercambio con el MENOR/IGUAL HIJO. Hago esto recursivamente.
- Bajar(p): Mientras que p no sea hoja y la prioridad de p sea menor al mayor de sus hijos, intercambio el hijo con p.

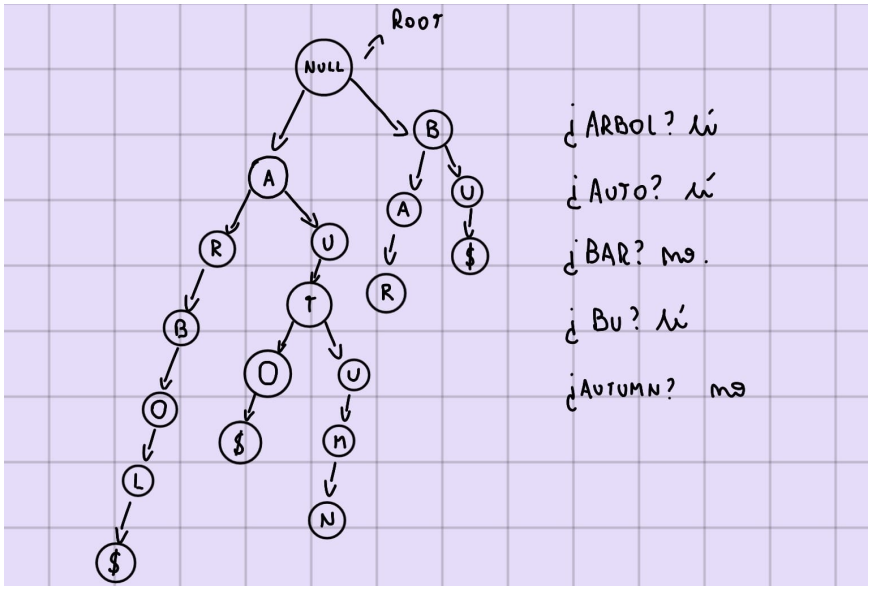
Transformaciones Array a Heap

Véase array → heap

Tries

El Trie es una estructura de datos que se suele utilizar en autocompletado de texto, pattern matching, etc.
La principal característica es que la complejidad depende del tamaño de los elementos (en general, palabras) y no de la cantidad de elementos.

- Se implementa sobre árboles n-arios: Donde n sería el tamaño del alfabeto más uno (por el símbolo reservado $)
- Se usan para guardar grandes conjuntos de palabras o secuencias de elementos con un alfabeto reducido
- Cada nodo representa un caracter o componente de los elementos guardados.
- Se marca con $ la finalización de la palabra o elemento. Donde el significado $ es el valor de la palabra entera.
  - El camino de la carrera Análisis I, en el valor de I el significado podría ser un objeto de esa materia.
- Los caracteres, se van colocando donde se desea completar la palabra, pero no puede tener un hijo derecho y sí un hijo izquierdo.



Propiedades:

- En el nivel i-ésimo del árbol se guarda en el i-ésimo componente de la palabra/secuencia.
- La estructura del trie es única, no importa el orden en que se ingresen los valores de los nodos.
- Complejidad o(m) donde m es la longitud de la cadena involucrada en el procedimiento.
- La raíz de todo Trie es nulo, porque es la palabra vacía.
- Las palabras, terminan con $. Si no hay un camino que termine con $ entonces no es una palabra válida.

Nota: Hay muchos algoritmos de Trie que tienen buena complejidad temporal pero son muy malos en complejidad espacial. Casos útiles para Trie:

- Si hay varias letras que se combinan y arman cosas diferentes, lo más seguro es que uses un trie para armar y buscarlas en O(—palabra—)
- Si hay varias letras que se combinan y arman cosas diferentes, lo más seguro es que uses un trie para armar y buscarlas en O(1), notése que acá están acotadas, por lo tanto no tenemos que hacernos drama porque nunca va a existir algo más grande.

Hashing

Refiere al proceso de tomar una entrada, de cualquier tipo y generar una cadena de caracteres de longitud fija que representa de manera única esta entrada.
Son adecuados para representar diccionarios y/o conjuntos.

Representación

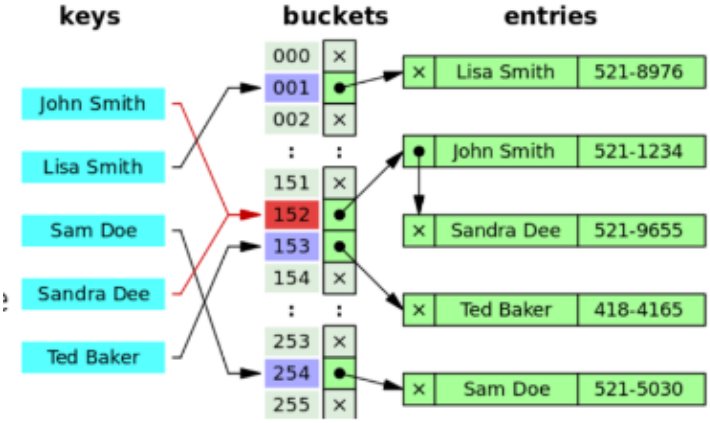
Representemos un diccionario con

- Tupla $< T, h >$ donde T es un arreglo con N = tamaño(T) celdas
- h *es una funcion hash* : $h(k) \to 0, ..., n-1$
  - k: Conjunto de claves posibles
  - 0, ..., n-1: Conjunto de las posiciones de la tabla (pseudoclaves)
  - Para poder conseguir la posición de un elemento en el arreglo usamos la función h.

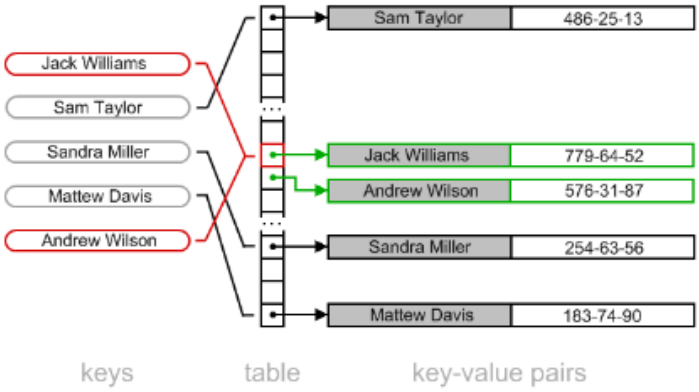
Colisiones

Las colisiones en un hash ocurre cuando dos entradas (keys) distintas producen el mismo valor de hash.
Un algoritmo de hash debería producir hashes únicos para cada entrada, sin embargo, debido a las limitaciones en la longitud finita de los hashes y a la naturaleza de la función hash es matemáticamente inevitable que ocurran colisiones en algún momento.
Un buen algoritmo de hash minimiza la probabilidad de colisiones
Maneras de resolver colisiones:

- Direccionamiento Cerrado / Encadenamiento: Cada celda de la tabla hash mantiene una lista enlazada de todos los elementos que han colisionado en esa celda. En el mejor caso, el facotr de carga α = cantidad de elementos/cantidad de posiciones. Las operaciones sobre la listas enlazadas tendrán $\Omega(N/|T|)$ y $O(N)$



- Direccionamiento abierto: Cuando ocurre una colisión, se busca otra ubicación vacía dentro de la tabla hash para almacenar el elemento adicional. Este método puede producir deslocalización del elemento ya que cuando vuelva a llamar a la clave, el valor de hash para esa clave no va a coincidir con la posición del elemento, sino que habrá que hacer el mismo recorrido a la hora de setearlo.



Barridos

También conocido como "scanning in hashing", se refiere a la acción de recorrer toda la tabla hash en búsqueda de un elemento específico o en caso de una colisión. Los barridos se diferencian por su complejidad algorítmica.

- Barrido lineal: Cuando ocurre una colisión el algoritmo busca secuencialmente la siguiente ubicación disponible en la tabla hash hasta encontrar una celda vacía.
  - Fórmula para encontrar la siguiente ubicación: $h(k,i) = (h'(k) + i)\ mod\ m$
  - h(k, i) = nueva función hash
  - h'(k) = función hash original
  - i = número de intentos de búsqueda
  - m = tamaño de la tabla hash
  - Los elementos se aglomeran por largos tramos y puede surgir **aglomeración primaria**.

- Barrido cuadrático: El incremento de las iteraciones es cuadrático.
  - Fórmula para encontrar la siguiente ubicación: $h(k,i) = (h'(k) + c_1 * i + c_2 * i^2)\ mod\ m$
  - El barrido cuadrático reduce el agrupamiento que a veces ocurre con el barrido lineal, sin embargo, puede sufrir de agrupamientos cuadráticos si no se eligen adecuadamente las constantes de incremento.
  - El agloramiento para este barrido se llama **aglomeración secundaria**
- Hashing doble: Se utiliza una segunda función de hash para calcular una ubicación alternativa cuando ocurre una colisión
  - Fórmula para encontrar la siguiente ubicación: $h(k,i) = (h_1(k) + i * h_2(k))\ mod\ m$
  - Donde $h_1(k)\ y\ h_2(k)$ son dos funciones de hash diferentes
  - i = número de intentos de búsqueda
  - m = tamaño de la tabla hash
  - El uso de una segunda función de hash reduce las posibilidades de agrupamiento y mejora la distribución de las claves en la tabla hash.
  - No se generan esas aglomeraciones o clusters tan frecuentes.

Requisitos para una función hash

- Determinismo: Dada una misma entrada, la salida debe ser siempre el mismo hash.
- Eficiencia: Debe ser rápido de calcular.
- Difusión: Pequeños cambios en la entrada deberían producir cambios significativos y difíciles de predecir en el valor hash.
- No reversibilidad: No se debe ser posible obtener el valor orignal a partir del hash.
- Unicidad: Cada entrada debería producir un hash único. Es decir, no debería haber dos entradas diferentes con el mismo hash (ej: uuid)
- Uniformidad de distribución: Los valores hash deberían estar distribuidos uniformemente a lo largo del espacio de salida posible. Esto mejora la eficiencia del hashing y reduce la probabilidad de colisiones

Anexo

Ejemplo 1 en cálculo de complejidad

```
1   void add(int mat1[n][n], mat2[n][n]){
2       int ans[n][n];
3           for(int i = 0; i < n; i++){
4               for(int j = 0; j < n; j = j+1){
5                   ans[i][j] = mat1[i][j] + mat2[i][j];
6               }
7           }
8       return ans[0][0];
9   }
```

Parte 1: Comenzamos colocando la cantidad de operaciones que se realizan, línea a línea. Los ciclos, en el caso del for, posee 3 casos: el caso inicial, caso intermedio, caso fin.

```
1   void add(int mat1[n][n], mat2[n][n]){
2   0     int ans[n][n];
3   2 2 2   for(int i = 0; i < n; i++){
4   2 3 3       for(int j = 0; j < n; j = j+1){
5   8               ans[i][j] = mat1[i][j] + mat2[i][j];
6               }
7           }
8   3     return ans[0][0];
9   }
```

Obs: j = j+1 ≠ j++ en temas de cantidad de operaciones. El j++ es una operación menos aunque hagan lo mismo. Obs: El return funciona como una asignación, por lo tanto también cuenta en operaciones.
Parte 2: Elegimos si empezamos por el mejor, o el peor caso. El peor caso suele ser el más tedioso así que empecemos por ese.
Parte 2.1: Peor caso
Parte 2.1.1 : Empezamos considerando el caso intermedio del ciclo.

```
2 3 3   for(int j = 0; j < n; j = j+1){
8           ans[i][j] = mat1[i][j] + mat2[i][j];
        }
```

$\sum_{j=0}^{n-1}(8+3)$
Ahora consideramos el caso inicial del ciclo, y el final: $2+\sum_{j=0}^{n}(8+3)+3$
Definimos: $a \equiv 2+\sum_{j=0}^{n-1}(8+3)+3$
Parte 2.1.2:

```
2 2 2   for(int i = 0; i < n; i++){
            a
        }
```

Comenzamos considerando el caso intermedio del ciclo. $\sum_{i=0}^{n}(a+2)$
Ahora consideramos el caso inicial del ciclo, y el final: $2+\sum_{i=0}^{n}(a+2)+2$
Definimos: $b \equiv 2+\sum_{i=0}^{n-1}(a+2)+2$
Parte 2.1.3: Juntar ambos ciclos $c \equiv 2+\sum_{i=0}^{n-1}((2+\sum_{j=0}^{n-1}(8+3)+3)+2)+2$
Parte 2.4: Resolver los ciclos, uniéndolos de alguna forma
$\equiv 2+\sum_{i=0}^{n-1}((2+\sum_{j=0}^{n-1}(\mathbf{8+3})+3)+2)+2$
$\equiv 2+\sum_{i=0}^{n-1}((\mathbf{2}+\sum_{j=0}^{n-1}11+\mathbf{3})+2)+2$
$\equiv 2+\sum_{i=0}^{n-1}((\sum_{j=0}^{n-1}\mathbf{11}+5)+2)+2$
$\equiv 2+\sum_{i=0}^{n-1}((11*\sum_{j=0}^{n-1}1+5)+2)+2$
$\equiv \mathbf{2}+\sum_{i=0}^{n-1}((11*n+5)+2)+\mathbf{2}$
$\equiv 4+\sum_{i=0}^{n-1}(11*n+5+2)$
$\equiv 4+\sum_{i=0}^{n-1}(11*n+7)$
$\equiv 4+n*(11*n+7)$
$\equiv 4+11n^2+7n$
Parte 2.1.5: Agregar el caso del return $\equiv 3+4+11n^2+7n$
$\equiv 11n^2+7n+7$
Parte 2.1.6: Observar cual es la variable con mayor exponente. En este caso, es $11n^2$ por lo tanto, $Tpeor(n) = \Theta(max\{11n^2, n, 7\})$.
Luego, $Tpeor(n) = \Theta(n^2)$
Nota: En la materia no utilizan los números de operaciones, pero acá por ejemplo 2, 8, 3, 3, 2 (operaciones elementales) serían $\Theta(1)$, osea nuestro cálculo de la Parte 2.1.5 se vería así: $\Theta(n^2)+\Theta(n)+\Theta(1)$
Parte 2.2: Mejor caso
El mejor caso es igual al peor caso porque no tenemos restricciones del ciclo más que las matrices de entradas y tampoco tenemos ninguna condición de corte. Siempre se recorre la misma cantidad de veces.

Ejemplo 2 en cálculo de complejidad

```
int russian(int a, int b){
    int res = 0;
    while(b > 0){
        if(b % 2 == 1){
            res = res + a;
        }
        a = a * 2;
        b = b/2;
    }
    return res;
}
```

A simple vista podemos notar algo, estamos usando un while y la guarda depende de un b que desconocemos. Veamos como va variando la variable b; Se observa que se va dividiendo por dos, entonces es algo más rápido que hacer b-1. Seguramente, el peor caso sea logarítmico (como en búsqueda binaria que se va partiendo la lista en dos)

Parte 1: Comenzamos colocando la cantidad de operaciones que se realizan, línea a línea.

```
1   int russian(int a, int b){
2       1     int res = 0;
3       1     while(b > 0){
4       2         if(b % 2 == 1){
5       2             res = res + a;
6                 }
7       2         a = a * 2;
8       2         b = b/2;
9             }
10      1     return res;
11  }
```

Parte 2: Elegimos si empezamos por el mejor, o el peor caso. El peor caso suele ser el más tedioso así que empecemos por ese.
Parte 2.1: Peor caso
Parte 2.1.1: Observamos que nuestro peor caso es que b > 0 y al dividir b por 2 sea impar.
Dentro del ciclo tenemos: (1 + (2+2) + 2 + 2) operaciones elementales.
Nótese que el 1 es de la condición de la guarda.
El ciclo es algo inverso, se comienza siendo algo muy grande y se va achicando, y el ciclo termina cuando b = 0. ¿Cuantas veces tengo que dividir b entre 2 para que podamos llegar a 0? El logaritmo base 2 de b, pero acá b lo llamamos n.
Parte 2.1.2: Utilizar logaritmo para hablar del ciclo
$\log(n)(1 + (2 + 2) + 2 + 2)$
Nótese que eliminamos la base 2 porque en complejidad, la base del logaritmo da igual.
Parte 2.1.3: Considerar la asignación inicial y el return junto al ciclo
$1 + \log(n)(1 + (2 + 2) + 2 + 2) + 1 \equiv 1 + \log(n)(8) + 1$
Parte 2.1.4: Calcular la complejidad $T_{peor}(n) = \Theta(1) + \Theta(log(n)) + \Theta(1) \equiv \Theta(max\{1, log(n), 1\}) \equiv \Theta(log(n))$
Parte 2.2: Mejor caso
Si b = 0 inicialmente, no entra nunca al ciclo por lo tanto no hay complejidad alguna porque todas son operaciones elementales.
$T_{mejor}(n) = \Theta(1) + \Theta(1) + \Theta(1) \equiv \Theta(max\{1, 1, 1\}) \equiv \Theta(1)$

Ejemplo 3 en cálculo de complejidad

```
1   function AlgoritmoQueHaceAlgo(arreglo A)
2   int i : = 1; int j := 1;
3   int suma := 1; int count := 0;
4   while i <= tam(A) do
5       if i != A[i] do
6           count : = count + 1;
7       end if
8       j := 1;
9       while j <= count do
10          int k := 1;
11          while k <= tam(A) do
12              suma := suma + A[k];
13              k := k * 2;
14          end while
15          j := j+1;
16      endwhile
17      i := i + 1;
18  endwhile
19  return suma
```

Lo primero que comenzamos viendo cual es el peor caso de las variables.
El ciclo más de adentro con guarda k depende del tamaño de A, por lo tanto el ciclo itera máximo $log(|A|)$ veces (xq k se va duplicando en cada caso).
El ciclo con guarda $j <= count$ depende de una variable count. En el peor caso ¿qué valor toma count? bueno, count se incrementa en el peor caso hasta ser $count = |A|$. Una observación importante es que count crece igual que —A— pero el proceso lo hace pasando por i = 0, i = 1 SIEMPRE.
El ciclo más de afuera con guarda i depende del tamaño de A, por lo tanto el ciclo itera máximo $|A|$ veces (xq empieza desde i = 1).

Luego, quedaría algo así:
$\Theta(1) + \Theta(1) + (\sum_{i=0}^{|A|} \Theta(1) + \Theta(1) + \Theta(1) + (\sum_{j=1}^{i} \Theta(1) + (\sum_{k=0}^{log(|A|)} \Theta(1) + \Theta(1)) + \Theta(1)) + \Theta(1)) + \Theta(1)$
$\equiv \Theta(1) + \Theta(1) + (\sum_{i=0}^{|A|} \Theta(1) + \Theta(1) + \Theta(1) + (\sum_{j=1}^{i} log(|A|) * \Theta(1)) + \Theta(1)) + \Theta(1)$
$\equiv \Theta(1) + \Theta(1) + (\sum_{i=0}^{|A|} \Theta(1) + \Theta(1) + \Theta(1) + (\sum_{j=1}^{i} log(|A|) * \Theta(1)) + \Theta(1)) + \Theta(1)$
$\equiv \Theta(1) + \Theta(1) + (\sum_{i=0}^{|A|} \Theta(1) + \Theta(1) + \Theta(1) + i * log(|A|) + \Theta(1)) + \Theta(1)$
$\equiv \Theta(1) + (\sum_{i=0}^{|A|} i * log(|A|)) \leftarrow$ Como i depende del ciclo, lo dejo adentro pero saco lo demás hacia afuera.
$\equiv \Theta(1) + (log(|A|) * \sum_{i=0}^{|A|} i)$
$\equiv \Theta(1) + log(|A|) * |A|(|A| + 1)/2$
$\equiv log(|A|) * |A|^2/2 + |A|/2$
$\equiv log(|A|) * |A|^2/2 + log(|A|) * |A|/2$
$\equiv log(|A|) * |A|^2 + log(|A|) * |A| \leftarrow$ Saco denominadores, son constantes.
$\equiv log(|A|) * |A|^2$
$\equiv \Theta(log(|A|) * |A|^2)$
Conclusión: Es importante revisar que si el ciclo depende de una guarda como $j \leq count$ revisar como va creciendo count; como count depende del valor de i, i va siendo 1, 2, 3, 4, osea que no sucede una sola vez y no podemos generalizar. El peor caso sería que i pase por 1, 2, 3, 4, hasta tam(A), entonces el ciclo que depende de count se ejecuta 1 vez, despues 2 veces, después 3 y así.

Ejemplo 3 en cálculo de complejidad

Recuerdo:

- $\Theta(n)$: Cota ajustada, exactamente n.
- $\Omega(n)$: Funciones mayores a n.
- $O(n)$: Funciones menores a n.
- c: Variable real mayor a 0. Nos basta con que exista una.
- n: Mayor a n0 donde ambas son naturales, donde n es suficientemente grande.

Determine la verdad o falsedad de las siguientes afirmaciones:

1. $2^n = O(1)$
Para empezar, la igualdad de una función con un grupo de funciones significa lo siguiente $2^n \leq c * 1$
Luego, $2^n = c$ es claramente es falso.
Contraejemplo: $n = 4, 2^4 \neq O(1)$

2. $n = O(n!)$
Esperaría que sea verdadero porque es cierto que el conjunto de funciones menores a n! incluye a n. $n \leq c * n!$
$n \leq c * (n-1)! * n$
$1 \leq c * (n-1)! \leftarrow$ trivialmente verdadero para n suficientemente grande y c mayor que cero.

3. $n + m = O(nm)$
Sabemos que n y m son dos valores de salida de una función. Parecería verdadero que una multiplicación de dos cosas incluya a la suma de ambas.
$n + m \leq c * nm$
$n/nm + m/nm \leq c$
$1/m + 1/n \leq c \leftarrow$ trivialmente verdadero porque un $c > 0$ siempre va a ser mayor a dos racionales que tienden a 0.

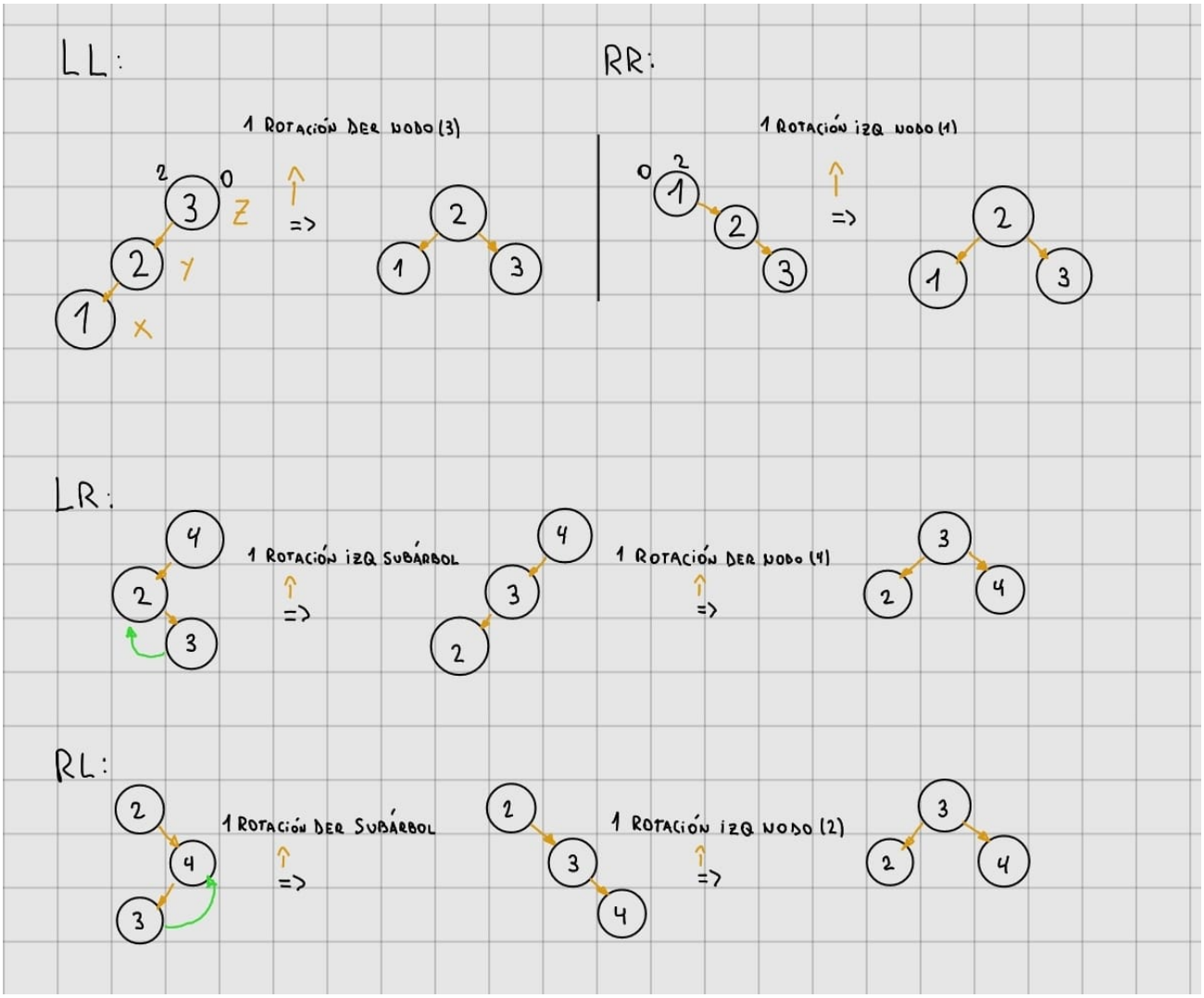
4. $f \in O(log\ n)\ entonces\ f \in O(n)$
Esto es trivialmente verdadero a la vista porque sabemos que si f es una función que es menor a O(log n) entonces obligatoriamente f es menor a O(n) por transitividad $\implies f \leq O(log\ n) \leq O(n)$

$f \in O(log\ n) \equiv f \leq c * log(n),\ f \in O(n) \equiv f \leq c * n$
$\implies f \leq c * log(n) \implies f \leq c * n$
$\implies c * log(n) \leq c * n$
$\implies log(n) \leq n \leftarrow$ trivialmente verdadero.

Single Linked List - Complejidad - Implementación

```
Struct NodoLista<T> {
    prev: NodoLista<T>
    sig: NodoLista<T>
    val: T
}

Módulo LinkedList<T> implementa Secuencia<T> {
    var primero: NodoLista<T>
    var ultimo: NodoLista<T>
    var longitud: int

    Costo peor caso / mejor caso = O(1) xq ya tengo primer puntero y lo peor que tengo que hacer es reordenar 1 solo.
    proc agregarAdelante(inout l: LinkedList<T>, in e: T) {
        var nuevoNodo: NodoLista<T> = new NodoLista<T>(e);
        var primeroAnt: NodoLista<T> = l.primero;
        nuevoNodo.sig = primeroAnt;
        l.primero = nuevoNodo;
        if(l.longitud == 0){
            l.ultimo = nuevoNodo;
        }
        l.longitud+=1;
    }

    Costo peor caso / mejor caso = O(1) xq ya tengo el último puntero y lo peor que tengo que hacer es reordenar 1 solo.
    proc agregarAtras(inout l: LinkedList<T>, in e: T){
        if(l.longitud == 0) then
            agregarAdelante(l, e);
        else
            var nuevoNodo: NodoLista<T> = new NodoLista<T>(e);
            var anteriorUlt: NodoLista<T> = l.ultimo;
            anteriorUlt.sig = nuevoNodo;
            l.ultimo = nuevoNodo;
        endif
    }

    proc buscar(in l: LinkedList<T>, in e: T): T {
        // Si e == l.primero.val -> costo O(1)
        //Si e NO es el primero, entonces -> costo O(n)

        Aunque tenga el último guardado, no puedo sacarlo como caso aparte porque para calcular la complejidad tendría que aíslar los casos del primero y el último y es un quilombo.
    }
}
```

Rotaciones según desbalance de AVL

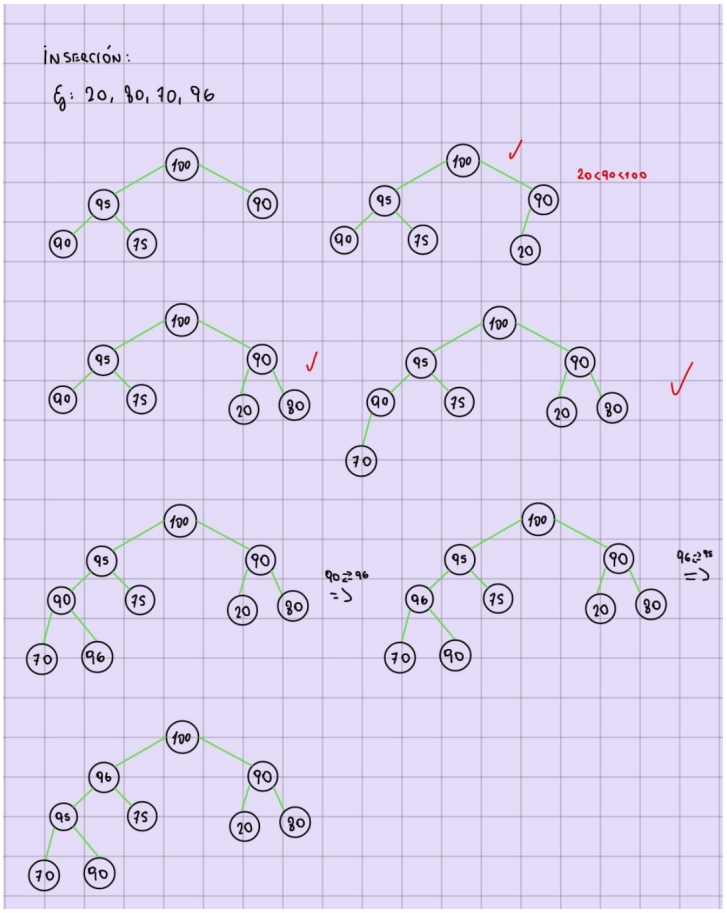


Elección de estructura para Cola de Prioridad

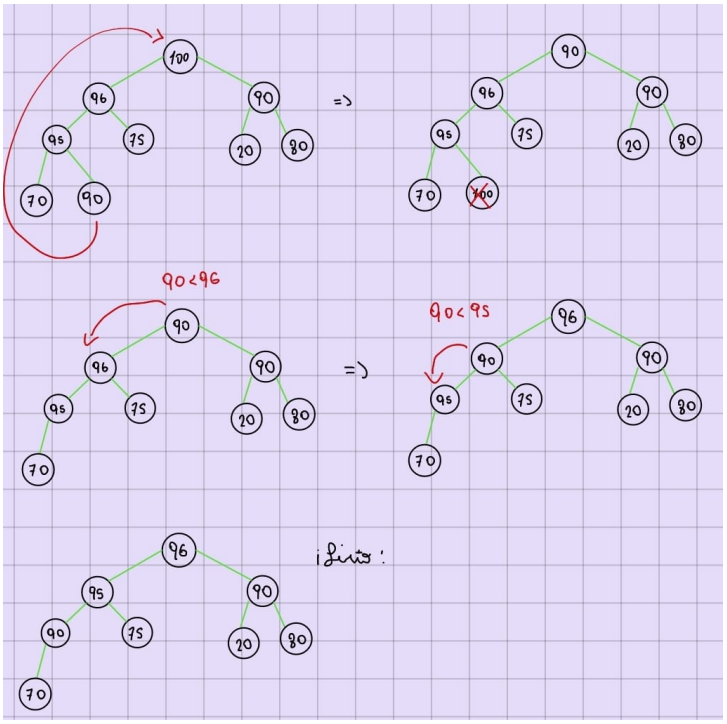
La mejor forma y más rápida de implementar una cola de prioridad sería con un árbol AVL. Esto nos garantiza que la búsqueda en el peor caso es O(log n).
Además, las inserciones podrían ser en tiempo logarítmico y los borrados también.
Una mejor solución y más elegante (que no nos mejora nada en temas de complejidad) podría ser usando Heaps.
Nota: El AVL es bueno cuando usamos diccionarios; pero si no usamos diccionarios para la cola de prioridad, en este caso es mejor usar Heaps.

Operaciones Heaps

Inserción:



Desencolar / Borrar:



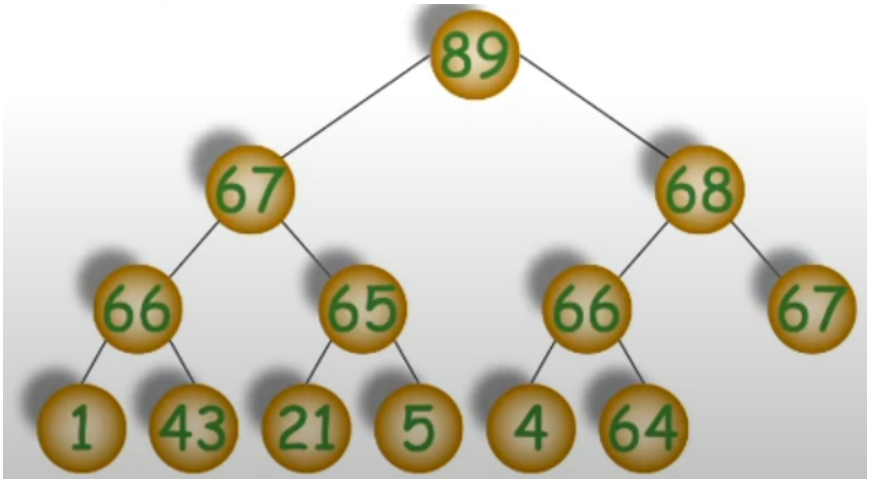
Nota: Nótese que es importante si está implementando un max-heap o un min-heap, porque en el max-heap tenemos que reacomodar de manera que quede en la raíz el más grande mientras que en el min-heap tenemos que dejar el más chico.

Implementacion de Heaps de maneras diversas

Sobre Array Sabemos que un Heap es lo más cercano a ser implementado con árboles binarios balanceados. Consideremos que debemos implementarlo con un array ¿como lo haríamos?

La manera de representarlo en un array es: raiz - hijo izq - hijo der - hijo izq de hijo izq, hijo der de hijo izq - hijo izq de hijo der, hijo der de hijo der.
Es como que vas tomando misma altura, pero completas primero con izquierda y luego derecha.
Fórmula (invariante):

- si v es la raiz, entonces v es el primero del arreglo.
- si v es el hijo izquierdo de u entonces 2p(u)+1
- si v es el hijo derecho de u entonces 2p(u)+2



Transformado a array sería: $[89, 67, 68, 66, 65, 66, 67, 1, 43, 21, 5, 4, 64]$
Ventajas de implementar Heap con Array:

- Ventajas:
  - Muy eficiente en términos de espacio.
  - Es fácil de navegar.
- Desventajas:
  - Al ser un arreglo, es necesario duplicar el arreglo o achicarlo a medida que se agregan o eliminan elementos.

Array → Heap (Array2Heap)

- Forma 1 (Costo $\theta(n\ log(n))$): Permutando sus elementos, hasta llevarlo a un heap y que cumpla el invariante de representación. Si no recuerda qué reglas debe cumplir el array, véase implementaciones heap.
- Forma 2 (Costo $\theta(n)$), Algoritmo de Floyd: Estrategia bottom-up. Desde el último sub-heap de la derecha, voy viendo si cada uno son un heap válido, si no lo son aplico la operación bajar en el elemento padre de ese sub-heap. Si ya llegué al último sub-heap de la izquierda, vuelvo a la derecha pero voy al padre del último padre de la derecha.