Trabajo Práctico 1

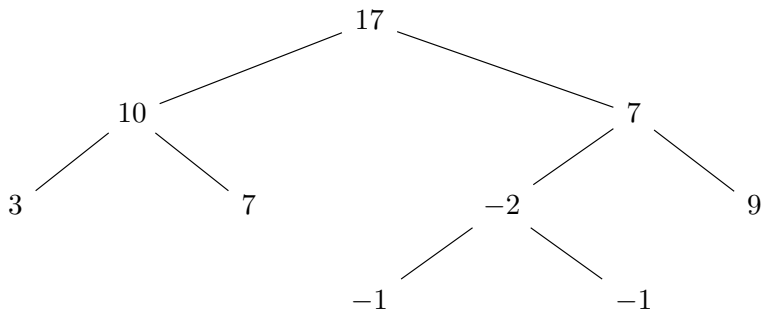
Introducción

Una cuerda o *rope* es una estructura de datos que permite representar secuencias de enteros y calcular la suma de subsecuencias de manera eficiente.

En concreto, una *rope* es un árbol binario donde

- las hojas almacenan los enteros que definen la secuencia y
- los nodos internos almacenan la suma de enteros correspondiente al subintervalo definido por sus hojas descendientes.

Por ejemplo, una representación gráfica de la secuencia $[3, 7, -1, -1, 9]$ es



Es interesante notar que la estructura es recursiva y por lo tanto los subárboles de la raíz de un *rope* son también un *rope*. También es posible combinar un par de *ropes* en tiempo constante con nueva raíz sumando los valores del par de raíces de *ropes* a combinar en (glorioso) tiempo constante.

Parte I

Operaciones sobre cuerdas

Actualización La actualización de un valor en la secuencia implica modificar una hoja y luego actualizar el camino de todos los nodos internos ancestros de dicha hoja hasta alcanzar la raíz.

Una implementación razonable actualiza el camino antes mencionado desde la hoja modificada hasta la raíz y lo hace en tiempo proporcional a la alura del árbol.

Suma de intervalo de valores Para calcular la suma de un intervalo $[l, r)$ distinguimos en casos. Si n representa un nodo interno del *rope* correspondiente a un intervalo $[l', r')$

- Si $[l', r') \subseteq [l, r)$ entonces se retorna la suma almacenada en n.
- Si $[l', r') \cap [l, r) = \emptyset$ se retorna 0.
- Si no, $[l, r)$ tiene intersección no nula con uno o ambos subárboles con lo cual se retorna la suma del inetrvalo en cada uno de los hijos de n.

Una implementación razonable calcula la suma de una subsecuencia en tiempo proporcional a la alura del árbol.

Nota Para cada nodo interno se debe registrar el intervalo de índices con el que éste se corresponde en la secuencia o alguna información equivalente.

Implementación en C++

La definición de la estrucura *rope* no exige ninguna condición de balanceo pero debería resultar evidente que un árbol no balanceado puede degenerar en una lista y por lo tanto los costos de las operaciones presentadas empeoran significaivamente.

Si sólo consideramos la clase de árboles completos, resultan éstos balanceados en altura y por tanto los costos se aproximan más a funciones logarítmicas que a funciones lineales. Más aún, los árboles completos admiten una representación en un arreglo que llamaremos árbol implícito.

Bajo esta representación

- una secuencia de N elementos tiene $2 * N - 1$ nodos de los cuales N son hojas y $N - 1$ nodos internos,
- el índice 0 se corresponde con la raíz del *rope*,
- para cada nodo de índice i su hijo izquierdo se encuentra en $2 * i + 1$ y su hijo derecho en $2 * i + 2$ y
- para cada nodo de índice i distinto de la raíz, su padre se encuentra en en $\lfloor \frac{i-1}{2} \rfloor$.

Para construir secuencias de largo distinto a una potencia de 2 resolvemos construir secuencias del menor largo potencia de 2 mayor o igual al largo de la secuencia y completamos las posiciones vacías con 0.

Tomando estas convenciones, podemos implementar la operacion de actualizacion de un *rope* sobre una secuencia de N elementos como sigue.

```
#define N 8
int valor[2*N+1];

// argumentos:
// nodo    - nodo a actualizar
// l_ y r_ - intervalo cubierto por nodo
// i       - indice a actualizar
// x       - nuevo valor a asignar
static void update_impl(int nodo, int l_, int r_, int i, int x) {
  int l = i, r = i+1;
  if (l <= l_ && r_ <= r) { valor[nodo] = x; return; }
  if (r <= l_ || r_ <= l) return;
  int m_ = (l_ + r_) / 2;
  // obs: una sola de estas dos llamadas hace algo, la otra cae en el caso base trivial
    inmediatamente
  update_impl(izq(nodo), l_, m_, i, x);
  update_impl(der(nodo), m_, r_, i, x);
  valor[nodo] = valor[izq(nodo)] + valor[der(nodo)]; // actualizo de abajo hacia arriba
}

void update(int i, int x) {
  update_impl(0, 0, N, i, x);
}
```

Nota No hace falta almacenar el intervalo comprendido por cada nodo, ya que lo podemos calcular a medida que avanza el algoritmo.

Estructuras genéricas

El lenguaje de programación C++ provee una forma de polimorfismo paramétrico (similar a las funciones polimórficas de Haskell) que permite implementar estructuras de datos genéricas de una forma similar a la que se ve en Estructuras de datos I en C, pero más elegante y con mayor seguridad de tipos.

Por ejemplo, podemos definir un arreglo dinámico que duplica su tamaño cuando está completo como sigue.

```
template<typename T>
struct DynamicArray {
  T* data = nullptr;
  int size = 0;
  int capacity = 0;

  void push(T x) {
    if (size == capacity) {
      int new_capacity = max(4, capacity * 2);
      T* new_data = new T[new_capacity];
      for (int i = 0; i < capacity; ++i) {
        new_data[i] = data[i];
      }
      delete[] data;
      data = new_data;
      capacity = new_capacity;
    }
    data[size++] = x;
  }

  T get(int i) {
    return data[i];
  }
}
```

para luego usarlo como

```
int main() {
  DynamicArray<int> arr;
  arr.push(1);
  arr.push(2);
  arr.push(3);
  arr.push("pepe"); // error de tipos: esperaba 'int' pero recibio 'const char*'
}
```

De cualquier manera, en el caso de arreglo dinámico tenemos disponible la implementación de la biblioteca estándar

```
#include <vector>
int main() {
  std::vector<int> arr = {1, 2};
  arr.push_back(3);
  arr.push_back("pepe"); // error
}
```

Polimorfismo restringido en C++

Al igual que Haskell tiene *class* para restringir funciones polimórficas, C++ tiene *concept*. Por ejemplo, podemos definir el concepto de un grupo como sigue.

```cpp
#include <concepts> // std::same_as

template<typename T>
concept Group = requires(T::Value a, T::Value b, T::Value c) {
    typename T::Value;                                      // hay un tipo de valores
    { T::op(a, b) } -> std::same_as<typename T::Value>;     // clausura de la operacion
    // T::op(a, T::op(b, c)) == T::op(T::op(a, b), c)       // asociatividad de la operacion
    { T::neut() } -> std::same_as<typename T::Value>;       // existencia del neutro
    // T::op(a, T::neut()) == a                             // neutro por derecha
    // T::op(T::neut(), a) == a                             // neutro por izquierda
    { T::inv(a) } -> std::same_as<typename T::Value>;       // clausura del inverso
    // T::op(a, T::inv(a)) == T::neut()                     // inverso por derecha
    // T::op(T::inv(a), a) == T::neut()                     // inverso por izquierda
};
```

Al igual que las leyes que definen muchas clases en Haskell, los requisitos definidos en comentarios no son chequeados por el compilador pero sirven como documentacion del uso esperado de este concept

Luego podemos escribir un tipo *cubierto* por ese concepto y una función polimórfica *restringida* a ese concepto.

```cpp
// Implementa el concepto de grupo
struct int_suma {
    using Value = int;
    static int op(int x, int y) { return x + y; } // una operacion asociativa
    static int neut() { return 0; }               // elemento neutro para op
    static int inv(int x) { return -x; }          // inverso para op respecto a neut()
};

template<typename G>
requires Group<G>
// sintaxis alternativa: template<Group G>
void invert_all(std::vector<typename G::Value>* arr) {
  for (int i = 0; i < arr->size(); ++i) {
    (*arr)[i] = G::inv((*arr)[i]);
  }
}

int main() {
        std::vector<int> v = {1, 2, 3};
        invert_all<int_suma>(&v);
}
```

Actividades

Ej. 1.

a) Implemente un *rope* con árbol implicito subyacente y operación suma. La función de inicialización debe recibir el tamaño de la secuencia como argumento.

b) Considere las siguienes variantes de *rope*: valores enteros con operación suma, valores *string* con operación concatenación y valores conjunto de enteros con operación unión.

¿Qué propiedades deben satisfacer los valores almacenados en las hojas de un *rope* para que se cumplan las propiedades vistas en la introducción y los costos sugeridos para las operaciones?

c) Implemente el *rope* como una estructura genérica en C++ restringida por un concepto adecuado, incluyendo las leyes necesarias escritas en comentarios.

d) Instancie y testee su implementación, al menos con la suma de enteros.

Parte II

Una extensión natural a la estructura de *rope* consiste en agregar la operación de actualizar valores de toda una subsecuencia. Esta operación suma una constante a todos los elementos de un intervalo $[l, r)$.

Una implementación directa consiste en modificar la función update_impl para que tome un intervalo $[l, r)$.

```
static void update_impl(int nodo, int l_, int r_, int l, int r, int x) {
  int len = r_ - l_;
  if (l <= l_ && r_ <= r && len == 1) { valor[nodo] += x; return; }
  if (r <= l_ || r_ <= l) return;
  int m_ = (l_ + r_) / 2;
  // obs: las dos llamadas pueden ser no-triviales simultaneamente
  update_impl(izq(nodo), l_, m_, i, x);
  update_impl(der(nodo), m_, r_, i, x);
  valor[nodo] = valor[izq(nodo)] + valor[der(nodo)]; // actualizo de abajo hacia arriba
}
```

Esta operación tiene costo lineal en el largo de la secuencia.

Actualización perezosa

Cuando una operación llega a su mínimo costo teórico, es común pensar en optimizaciones que mejoren el desempeño al considerar una conjunto de aplicaciones de la operación y no una aplicación individual. Éste análisis de costo se conoce como análisis de costo amortizado.

En lugar de actualizar los nodos hasta las hojas, podemos, una vez que el intervalo de la actualizacion cubre un nodo, actualizar ese nodo y dejar todos sus descendientes sin actualizar aunque dejando un marcador que indique que deben ser actualizados mas adelante.

Esto es posible de hacer eficientemente ya que es fácil calcular de qué manera se modifica la suma de un intervalo si se incrementan todos los valores de dicho intervalo en una constante.

Además, si hay nodos con actualizaciones pendientes resulta sencillo ver cómo se combinan las actualizaciones pendientes con las nuevas actualizaciones.

Implementado correctamente, este algoritmo logra costo logarítmico en cada actualización.

```
int lazy[2*N-1];
int valor[2*N-1];
void propagate(int node, int l_, int r_) {
    int len = r_ - l_;
    if (len > 1) { // no es hoja, combino actualizaciones en los hijos
        lazy[izq(nodo)] = lazy[izq(nodo)] + lazy[nodo];
        lazy[der(nodo)] = lazy[der(nodo)] + lazy[nodo];
    }
    valor[node] = valor[node] + lazy[node] * len;
    lazy[node] = 0;
}
void update_impl(int node, int l_, int r_, int l, int r, int upd) {
```

```
    propagate(node, l_, r_);
    if (l <= l_ && r_ <= r) { lazy[node] = upd; propagate(); return; }
    if (r <= l_ || r_ <= l) { return; }
    int m_ = (l_ + r_) / 2;
    update_impl(izq(node), l_, m_, l, r, upd);
    update_impl(der(node), m_, r_, l, r, upd);
    valor[nodo] = valor[izq(nodo)] + valor[der(nodo)];
}
```

Actividades

Ej. 2.

a) Implemente un *rope* con actualización de sumar en un intervalo y consulta de suma de un intervalo.

b) Considere tambien la variante donde la consulta obtiene el máximo elemento de un intervalo y la actualizacion suma una constante en un intervalo.

¿Qué propiedades deben tener el conjunto de valores y el conjunto de actualizaciones para poder actualizar perezosamente?

En particular, para poder combinar actualizaciones pendientes con nuevas actualizaciones y para poder calcular el resultado de aplicar la operacion perezosa a un intervalo sin tener que recursionar.

c) Implemente un *rope* con actualizaciones en intervalo genérico, definiendo un *concept* adecuado.

d) Instancie y testee su implementación genérica de *rope*, probando al menos las dos operaciones propuestas anteriormente.