Trabajo Práctico Final:
File System y Árboles de prefijos

Estructuras de Datos
Tecnicatura en Programación Informática
Universidad Nacional de Quilmes

1. Introducción

Un *árbol de prefijos* (o *trie*, por *re**trie**val* en inglés), es una estructura de datos pensada para representar eficientemente *mappings* en donde las claves de búsqueda son típicamente *strings* o, lo que es equivalente, listas de caracteres. Dado que estas listas pueden llegar a ser arbitrariamente grandes, se buscar evitar guardar la clave correspondiente a cada valor en su respectivo nodo del árbol. En cambio, la posición del nodo dentro de la estructura es la que determinará a qué clave está asociado su valor. Como consecuencia, se evita no solo guardar varias veces el mismo prefijo común a varias claves distintas, sino también realizar comparaciones de complejidad lineal (en el tamaño de las listas) por cada nodo que interviene en una búsqueda.

Continuando la línea del último parcial, este trabajo consistirá en implementar un *File System Simple*, haciendo uso tanto del tipo abstracto `Disc` (visto en dicho examen) como de un *Trie* para almacenar la información correspondiente a los distintos archivos. La implementación deberá ser imperativa y en el lenguaje C.

2. Árbol de prefijos

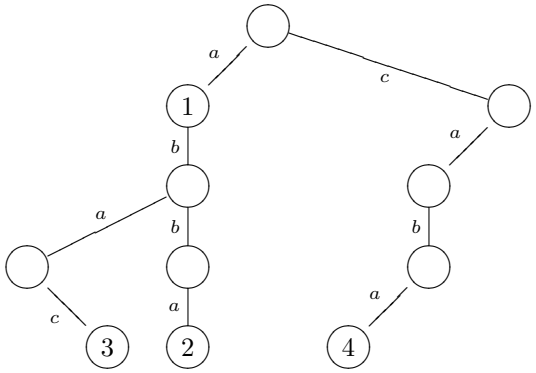
El tipo abstracto a implementar usando *árboles de prefijos* no es otro que `Map`, asumiendo que las claves son listas de `char` (se requiere usar el tipo `List` visto en clase). La interfaz del TAD es la siguiente:

- `Map emptyM();`
  Retorna el mapping vacío. $O(1)$
- `void assocM(Map &m, CharList k, MAP_ELEM_TYPE v);`
  Asocia la clave `k` al valor `v` en el mapping `m`. $O(k)$
- `void deleteM(Map &m, CharList k);`
  Elimina el valor asociado a la clave `k` en el mapping `m`. $O(k)$
- `Maybe lookupM(Map m, CharList k);`
  Retorna el valor asociado a la clave `k` en el mapping `m`, si éste existe. $O(k)$

Fijado un alfabeto de n caracteres, un trie es un árbol n-ario en donde, dado un nodo, cada uno de sus hijos representa un caracter dentro del alfabeto. De este modo, las claves quedan representadas por los distintos caminos en el árbol. A su vez, cada nodo que corresponda a una clave definida en el mapping, contiene su valor asociado.

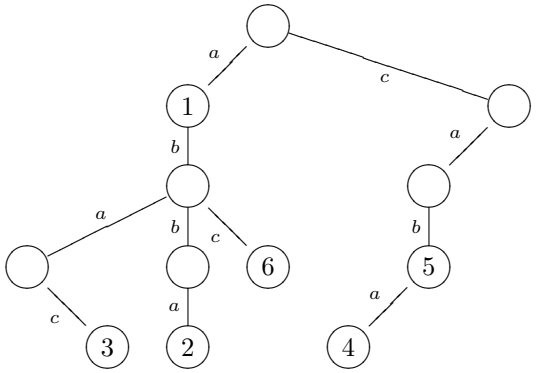
Para ilustrar esta lógica asumamos un alfabeto fijo de tres caracteres {`a,b,c`}. El mapping resultante de asociar los siguientes pares clave–valor: `("a":1)`, `("abba":2)`, `("abac":3)`,

`("caba":4)`; queda representado por el árbol:

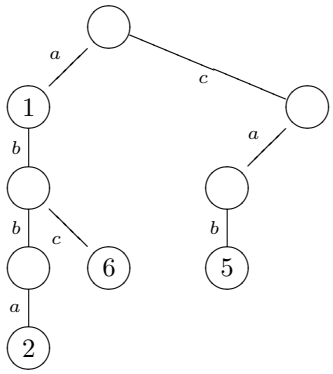


Notar que *no todos* los nodos guardan valores, sino solo aquellos correspondientes a prefijos que están efectivamente definidos en el mapping. Además, las hojas del árbol *siempre* contiene un valor.

El proceso de inserción consiste de seguir el prefijo correspondiente a la nueva clave mientras éste se encuentre definido en el árbol. Si la nueva clave es un prefijo de una ya definida, simplemente se inserta su valor asociado en el nodo correspondiente (ej. `("cab":5)`). Sino, se debe completar con los nodos necesarios hasta consumir la clave completa e insertar el valor en la nueva hoja creada: (ej. `("abc":6)`):



El proceso de eliminación puede resultar un poco más complejo, ya que se debe garantizar que no quedan hojas sin valores. Ésto quiere decir que, al eliminar un valor que se encontraba en una hoja, se debe seguir eliminando hacia arriba en el árbol el sufijo correspondiente a la clave, hasta encontrar un nodo que forme parte de otro prefijo (ej. `"caba"`). Notar que el nodo perteneciente a otro prefijo no necesariamente contiene un valor (ej. `"abac"`):



Ejercicio 1

Implementar el tipo abstracto de datos `Map` sobre árboles de prefijos, definiendo los invariantes

correspondientes. El alfabeto usado debe incluir al menos las 26 letras del abecedario en minúscula (a–z, evitando la $\tilde{n}$).

3. File System

Nuestro *File System Simple* (`FSS`) consistirá de dos estructuras que deben estar sincronizadas en todo momento. Por un lado se cuenta con un `Disc` que se encargará de administrar los bloques que se asignan para cada archivo creado. Por otro lado se tendrá un `Map` que contendrá la información correspondiente a cada archivo.

Recordemos la interfaz de `Disc`, donde `Block` es un simple renombre de `int`:

- `Disc nuevo(int n);`
  Genera un disco vacío de tamaño `n`. $O(1)$
- `int tamano(Disc d);`
  Retorna el tamaño de `d`. $O(1)$
- `bool libre(Block b, Disc d);`
  Decide si el bloque `b` está libre en el disco `d`. $O(log(d))$
- `bool ocupado(Block b, Disc d);`
  Decide si el bloque `b` está ocupado en el disco `d`. $O(log(d))$
- `void liberar(BlockList bs, Disc &d);`
  Libera todos los bloques de la lista `bs` en `d`. $O(log(d))$
  PRECONDICIÓN: los bloques de `bs` están ocupados en `d`.
- `BlockList ocupar(int n, Disc &d);`
  Modifica el disco `d` ocupando `n` bloques y retorna la lista de bloques usados. $O(log(d))$
  PRECONDICIÓN: hay el menos `n` bloques libres en `d`.
- `int espacioLibre(Disc d);`
  Calcula el espacio libre disponible en `d`. $O(d)$

Estas dos estructuras (disco y mapping) deberán mantener la consistencia de modo que los bloques ocupados en el disco sean exactamente los utilizados para los archivos.

Por simplicidad, de cada archivo nos interesa saber solo su nombre, tamaño y los bloques que lo componen. Se deberá definir una estructura `FileInfo` adecuada para preservar esa información en la representación de `FSS`.

La interfaz del TAD `FSS` es la siguiente:

- `FSS montar(int n);`
  Monta un file system con un disco de tamaño `n`. $O(1)$
- `int tamanoDisco(FSS fs);`
  Retorna el tamaño de `d`. $O(1)$
- `bool existe(CharList fn, FSS fs);`
  Decide si el archivo de nombre `fn` ya existe en el file system `fs`. $O(fn)$
- `void crear(CharList fn, int n, FSS &fs);`
  Crea un archivo de nombre `fn` y tamaño `n` en `fs`. $O(log(d) + fn + n)$
  PRECONDICIÓN: hay espacio suficiente en el disco `d` del file system.
- `BlockList abrir(CharList fn, FSS fs);`
  Retorna la lista de bloques correspondiente al archivo `fn`. $O(fn)$
  PRECONDICIÓN: el archivo `fn` existe en `fs`.

- `void borrar(CharList fn, FSS &fs);`
  Elimina el archivo `fn` de `fs`. $O(log(d) + fs + n)$
  PRECONDICIÓN: el archivo `fn` existe en `fs`.
- `int espacioDisponible(FSS &fs);`
  Calcula el espacio libre disponible en `fs`. $O(1)$

Ejercicio 2

Implementar el tipo abstracto de datos `FSS` utilizando al menos un `Map` (con `MAP_ELEM_TYPE = FileInfo`) y un `Disc` para representarlo. Definir los invariantes correspondientes.

— ¡Mucha suerte! —