Estructuras de Datos Avanzadas:

Tarea 3

Aplicaciones de las estructuras de datos

M. CONCHA VÁZQUEZ

Facultad de Ciencias, UNAM

Facultad de Ciencias
Universidad Nacional Autónoma de México
http://www.fciencias.unam.mx/

Título del proyecto:
Tarea 3

Tema del proyecto:
Estructuras de datos; estructuras de datos avanzadas; pilas; STACKS; *skip lists*; *skip graphs*; *árboles de búsqueda balanceados*; *BSTs*; árboles rojo-negros; árboles *splay*; DCEL; mapas trapezoidales; estructuras de datos geométricas; árboles Kd; árboles de segmentos; *min-max heaps*; *heaps* binomiales; *tries*; árboles de sufijos.

Periodo:
Diciembre, 2018.

Número de Grupo:
7093

Participantes:
Concha Vázquez Miguel.

Supervisores:
Profesora Adriana Ramírez Vigueras.
Luis Enrique Chacón Ochoa.

Fecha en que se completó el proyecto:
14 de diciembre del 2018.

Resumen:

Se detallan diversas aplicaciones que se han encontrado a algunas de las estructuras de datos revisadas a lo largo de las unidades en el curso. Algunas de las aplicaciones tienen que ver con particularizaciones y la especialización o hibridización de las estructuras con algunas otras y van desde aplicaciones en redes distribuidas hasta la compresión de datos. Se pretende con este trabajo ilustrar el poder de las estructuras vistas en clase, entendiendo que pueden y son usadas actualmente en muchos ámbitos. El objetivo es el de informar al lector, además de motivarlo al estudio y revisión de estos temas con tal de poder llevar a cabo un análisis adecuado de la complejidad de las operaciones —en tiempo de ejecución y de espacio— para poder eventualmente pensar en nuevas aplicaciones o diseñar nuevas estructuras de datos.

Índice general

Capítulo 1

Estructuras de datos elementales

1.1. Pilas

A continuación describiré varias aplicaciones que tienen las pilas (*stacks*), pues a pesar de ser estructuras de datos relativamente simples en contraste con aquellas que revisamos ya más adentrados en el curso, no carecen de una enorme utilidad en varias áreas, sobre todo relacionadas con el diseño y algoritmos de los sistemas operativos y de la construcción de compiladores de acuerdo a lo expuesto en [18] y [19]. Exponen ambos autores que las pilas se han empleado desde hace mucho tiempo en cuatro áreas principales de la computación: la evaluación de expresiones, almacenamiento de la dirección de retorno (RETURN ADDRESS) de subrutinas, almacenamiento dinámico de variables locales y paso de parámetros de subrutinas. Ahondemos ahora en estas.

1.1.1. Pila de evaluación de expresiones

Se trata en realidad de una de las primeras aplicaciones importantes que tuvieron las pilas en relación al *hardware* y manejo de información en las computadoras. Consiste en ir llevando un registro de las etapas intermedias en la evaluación de expresiones y sirve también para respetar la precedencia de operadores en las expresiones aritméticas,labor íntimamente relacionado con el trabajo de los compiladores. Existen dos vertientes en las estrategias para el uso de este tipo de pilas, dependiendo de si el lenguaje es compilado o interpretado:

- En los lenguajes compilados, el *hardware* utiliza una sola pila para que el compilador pueda llevar un control de las operaciones pendientes en la generación de las instrucciones y pueda consular los resultados intermedios que van siendo acumulados en esta pila.
- En los lenguajes interpretados, que van siendo analizados instrucción por instrucción, se utilizan ahora dos pilas de evaluación de expresiones. En la primera se tienen aquellas operaciones que requieren de los resultados de otras operaciones de mayor precedencia para poder ser evaluadas en su totalidad. La segunda pila contiene solamente las entradas asociadas o correspondientes a las demás operaciones pendientes.

Para ejemplificar esta aplicación directa de las pilas, consideremos por un momento de la siguiente expresión:

$$X = (A/Y) + (B * Z)$$

En este caso, primero deberíamos evaluar el resultado de la primera expresión parentizada, esto es, hacer la división de A entre Y. Luego de llevar a cabo el cociente, se hace un *push* del resultado en la pila de evaluación de la expresión. Posteriormente, se lleva a cabo el producto de los factores B y Z y otra vez se hace un *push* de este nuevo resultado. Por último, se hace un *pop* dos veces para tenes acceso a estos dos resultados intermedios y poder realizar la adición de ambos y así poder guardarlo en X.

Como puede notarse, la aplicación de la pila permite manejar eficientemente múltiples niveles de precedencia para los operadores. Recomiendo el problema de programación de la sección de *Pilas y Colas* de *Interviewbit* en caso de querer profundizar en esta aplicación.

En caso de que la notación fuera prefija o posfija, la aplicación de la pila sigue siendo una opción ideal. Consideramos por ejemplo el caos de la notación posfija en donde el operador es colocado luego de los dos operandos. El pseudocódigo del algoritmo para la evaluación de las expresiones utilizando la pila sería:

Algoritmo para la evaluación de expresiones

```
p = crearPila()
while(entradaNoTodaLeida())
        token = obtenerSiguienteToken()
    if(esOperando(token))
        p.push(token)
    else if (esOperador(token))
        op2 = p.pop()
        op1 = p.pop()
        resultado = calc(token, op1, op2)
        p.push(resultado)
return p.pop()
```

1.1.2. Pila de direcciones de retorno

Una técnica que era usada antes de 1950 para guardar la dirección de la rutina a la que tendría que regresar la ejecución de un programa luego de la llamada a una subrutina (en lenguajes como `FORTRAN`, por mencionar a un exponente) era la de guardarla en un espacio reservado para el cuerpo de la subrutina en cuestión. Esto por supuesto impedía que la misma subrutina se llamara a sí misma, pues se ocasionaría que la dirección de retorno previamente gaurdada para regresar a la función o rutina original fuera perdida por completo. Debido a la naciente necesidad de permitir código reentrante, se llegó a la conclusión de que era trascendental que la direccfión de retorno fuera alojada de forma dinámica.

Para afrontar este nuevo problema se decidió usar una pila para ir guardando las direcciones de retorno, de forma tal que cada vez que se invoca o llamada a una subrutina, la dirección de retorno del programa o rutina que la llama es guardada (con un *push*) en la pila. Ahora, cada vez que una nueva subrutina era llamada —inclusive en las llamadas recursivas—, se aseguraba que el manejo automático de esta nueva pila permitiera regresar a la dirección

de las rutinas invocadoras sin ningún inconveniente.

Los autores mencionan de igual forma que en la mayoría de las computadoras de la actualidad se utiliza *hardware* específico para dar soporte a la pila de direcciones de retorno, como suele ser el uso de un apuntador a la pila (*stack pointer*) y la pila suele guardarse en una porción de memoria que no es para nada usada por el programa que hace referencia a ella.

1.1.3. Pila de variables locales

Antaño, lenguajes como `FORTRAN` la información de las subrutinas era manejada al asignar un espacio de memoria permanente para su código, manejando así lo relacionado con las variables locales que eran usadas por estas; no obstante, esto no permitía el código recursivo. En contraparte, una aplicación clave de las pilas es su uso para manejar las variables locales de las subrutinas y funciones, permitiendo la reentrancia al habilitar que el mismo código pueda ser usado por diferentes hilos (*threads*) de control.

Para poder manejar las variables estáticas locales dentro de un procedimiento apropiadamente en esquemas de multi-hilo se tiene que ser extremadamente cuidadosos. Esto se debe a que los valores pueden ser corrompidos o cambiados por otro hilo de ejecución, ocasionando que otro hilo trabaje con valores distintos de las variables locales del código en cuestión. La solución que se encontró fue la de reservar espacio en una pila de variables locales para que en cada llamada a la subrutina se estén alojando nuevos bloques de memoria en donde se guardan las variables locales que corresponden al hilo que hizo la invocación a la subrutina; durante toda su ejecución, el hilo propicia entonces que la función o subrutina esté trabajando y actualizando los valores de las variables locales del bloque de la pila que le corresponde y fue reservado inicialmente.

Otra de las ventajas que presenta este inteligente aplicación de las pilas es la economizar el uso de la memoria. La razón es que en este esquema del uso de la pila de variables locales, el espacio de la pila está siendo reusado por las subrutinas en tanto estas se van llamando, pues en el transcurso de la ejecución del programa, le profundidad de la pila varía dependiendo de los *pushes* con nuevas llamadas y los *pops* cuando termina al ejecución de una subrutina. En contraparte directa —en caso de no usarse esta estrategia—, se tendría que usar espacio para las variables locales de toda subrutina en todo momento independientemente de que la subrutina estuviera activa (en ejecución) o no fuera el caso.

1.1.4. Pila de parámetros

En realidad, siempre ocurre que cuando se llama a una subrutina, a esta se le deben proporcionar los parámetros con los cuales trabajará. Podría pensarse en primera instancia que dichos parámetros podrían ser guardados en los registros del procesador antes de la ejecución de los procedimiento, pero esto tiene el inconveniente de que al ser limitados, entonces se impondría una restricción en términos del máximo número de parámetros con los que podrían actuar las subrutinas.

Una solución sumamente usada en los sistemas operativos es el de pasar estos parámetros al copiarlos —o bien copiar simplemente apuntadores a los estos— dentro de una lista en la memoria de la rutina que llama a la subrutina. De nueva cuenta existe el problema bajo este esquema de que los mecanismos d reentrancia y recursión no serían posibles. Para darle la vuelta al problema se prefiere seguir la estrategia más flexible consistente en colocar

o copiar los parámetros dentro de una pila de parámetros antes de que se llame a la subrutina, de forma tal que en su ejecución, la subrutina llamada pueda hacer referencia al contenido que fue colocado encima de la pila y poder actuar conforma a ello, permitiendo ahora sí la recursión en los programas.

Nosotros pudimos evidenciar este uso en nuestro curso de Sistemas Operativos cursada con José David Flores Peñaloza, en donde en la práctica 5 tuvimos que implementar justamente la colocación de los parámetros de las funciones en la pila de parámetros para poderlas llamar sin problema alguno. Esto fue al trabajar con `PintOS`, un sistema operativo desarrollado por la Universidad de Stanford con propósitos ilustrativos para la plataforma *x86* y lanzado en el año de 2004.

1.2. *Skip-Lists*

En el curso revisamos en las primeras semanas esta estructura basada en sucesiones de listas ligadas en diferentes niveles, en donde cada nivel tenía cada vez un menor número de nodos y permitía realizar consultas eficientemente al permitir *atajos* o *saltos* en relación al recorrido de la lista completa. En [30] se describe una variante de esta estructura, basada en las mismas ideas y que tiene interesantes aplicaciones en el día a día: las *skip graphs*.

En la actualidad se ha visto un rápido crecimiento del Internet, redes sociales y cómputo en la nube, lo que ha propiciado que las personas estén cada vez más conectadas. De inmediato surgió la pregunta de cuál sería la manera más eficiente de poder guardar toda esa información en forma de gráficas, preservando la necesidad de tener un rápido a acceso a la información de la red cuando así fuera necesario. Por supuesto, para estos órdenes de magnitud en cuando a los tamaños de estas redes, representar las gráficas por medio de matrices de adyacencias no es para nada una buena idea.

Por ende, Shalani y Amritpal introdujeren las *skip graphs*, estructuras que son una particularización de las *skip lists* vistas en el curso y que tienen un uso importante en los sistemas y ambientes distribuidos como son las redes entre pares o iguales (*peer-to-peer, P2P*) y presentan propiedades valiosas como son la tolerancia a fallos y un buen balance de carga.

Están compuestas de una torre de listas doblemente ligadas en donde los elementos son almacenados en distintos nodos que pueden *caerse* en cualquier momento; la información es así distribuida en un gran número de nodos. A pesar de estos problema que son inherentes a las redes de computadoras y las redes distribuidas, la estructura permite que los nodos fallen sin perder ningún tipo de conectividad gracias a su tolerancia a fallos, además de que las inserciones, búsquedas y reparación de errores en la estructura (producto de errores de la red) pueden ser llevados a cabo de forma muy directa con algoritmos eficientes. Ha sido tal su éxito a partir de la aplicación de los conceptos detrás de las *skip lists* que llevaron a su concepción, que *skip graphs* maś específicas ya han sido estudiadas como son las *skip nets*, *skip webs* y las *rainbox skip graphs*.

Aquí, los nodos representan un recurso de la red que se necesita encontrar, en donde un nodo x tiene dos campos:

- Una llave, que puede ser el nombre del recurso y es arbitraria. Pro curstiones de notación se suelen represetnar con enteros $1, 2, \ldots, n$.
- Un vector de pertenencia $m(x)$ (*membership vector*) que sería ya el recurso.

Más aún, los algoritmos para este tipo de estructura particular con usos tan importantes tiene el beneficio de que varias operaciones son sumamente parecidas a las de las *skip lists*. Por ejemplo. la operación de búsqueda es igual con la adaptación de que ahora debe ejecutarse en un sistema distribuido. El proceso de búsqueda es así iniciado en el nivel de más encima del nodo que está buscando una llave y continúa hacia abajo si así lo requiere hasta llegar al nivel 0. Eventualmente se tiene que regresar la dirección del nodo que guarda la llave que había sido buscada o bien la dirección del nodo que contiene:

- La llave más grande que no excede la llave buscada.
- O bien la llave más pequeña que es mayor a la llave que había sido buscada, dependiendo de la implementación concreta.

Gracias a su propiedades de tolerancia a fallos [13], aleatoriedad en los fallos (caso dos terceras partces de los nodos deben fallar para ocasionar que la mayoría de los nodos ya no permanezcan en la componente principal), el tiempo de ejecución en promedio del orden $O(\log n)$ para las búsquedas y su buen balance de carga ha dado lugar a que esta estructura que puede pensarse esencialmente como una *skip lists* distribuida pensada para las redes tenga varias aplicaciones importantes en este ámbito como son:

1.2.1. Redes de pares (*P2P*)

Las redes P2P son cada vez más populares; aquí no hay como tal servidores ni clientes fijos, sino que todos los nodos de la red distribuida superpuesta (*overlay network*) pueden intercambiar directamente la información entre sí, actuando quizás al mismo tiempo como clientes y servidores. Debido al incremento en el uso de este tipo de redes lógicas alámbricas e inalámbricas, no es inusual que su número de usuarios alcance millones y por supuesto comienza a ser de interés la expansión y escalabilidad para que las búsquedas por los recursos y actualizaciones en su topología (los participantes pueden estar frecuentemente conectándose y desconectándose) sea eficiente. Además, al ser casi la norma de que se tengan fallos impredecibles. los mecanismos para manejar este tipo de perturbaciones debe ser bueno. Autores en [32] han informado a profundidad cómo estas estructuras basadas en *skip lists* y *skip graphs* de distintos tipos pueden ser usadas para representar la información contenida en los nodos de la red distribuida para lograr estos valiosos objetivos Se llegó a la conclusión de que los sistemas distribuidos y de multiprocesadores multihilo suelen ser los mejores para que se tengan mejores resultados al usar estas estructuras.

1.2.2. Cómputo en la nube

Ocurre también que los servicios en la nube son cada vez más usados y en estos ambientes los datos son accesados y actualizados de forma puramente distribuida. En consecuencia, las estructuras de datos distribuidas para el almacenamiento dinámico de los datos en este tipo de aplicaciones requieren de las propiedades de persistencia y autenticación, esto con tal de validar los datos recibidos y hacer que los nuevos cambios en los datos no hagan perder datos previos que puedan ser requeridos por otros usuarios del ecosistema distribuido. En [29] también se pone en evidencia con experimentos de gran escala que estas estructuras tienen un gran desempeño al ser usadas para este tipo de ambientes de cómputo en la nube

1.2.3. Servicios web y aplicaciones misceláneas

Es fundamental que en el mundo moderno se pueda dar de forma rápida el descubrimiento en la red de los servicios web capacees de proveer servicios particulares por medio del descubrimiento de los servicios en la red (*Information Retrieval, IR*). Este aspecto es de interés principal en áreas como servicios de comercio en línea, ciencia y comunidades de investigación de áreas de todo tipo; se ha demostrado que las *skip graphs* pueden ser empleadas para estos propósitos al modelas las redes eficientemente con los recursos web contenidos en los nodos, permitiendo que el proceso de descubrimiento sea rápido en [14].

También es importante hacer énfasis en que estas estructuras permiten hacer consultas y actualizaciones sobre datos ordenados, lo que tiene entonces aplicaciones en bases de datos de ADN, servicios basados en localizaciones y búsquedas de nombres de archivos y títulos de datos por medio de prefijos ara facilitar un rápido intercambio de mensajes. Básicamente, como los autores exponen en [30], las aplicaciones son vastas y aún falta mucho por explorar en relación a las *skip lists* y particularizaciones como las *skip graphs*, sobre todo hoy en día en que los ambientes distribuidos son cada vez más importante en este mundo tan conectado.

Capítulo 2

Árboles balanceados de búsqueda

2.1. Árboles Rojo-negros

2.1.1. *Kernel* de `Linux` y calendarizador *CFQ*

En la fuente [4] se explica la forma en que además de contar con una implementación de los árboles *radix*, el *kernel* de `Linux` cuenta con una implementación de los árboles rojo-negros, conocida dentro del núcleo como *rbtrees*. Se eligió este árbol porque no se quería que de pronto se tuviera una mayor latencia por el rebalanceo impredecible que pudiera presentarse con otro tipo de estructura arbórea.

El proceso de usar los árboles rojo-negros en este *kernel* se inicia al incluir el archivo de cabecera `< linux/rbtree.h >` y se menciona que Andrea Arcangeli, quien llevó a cabo la implementación, tuvo que tener mucho cuidado al decidir cómo se harían las comparaciones de los tipos arbitrarios que podrían soportar estos árboles para el *kernel*, sobre todo teniendo en mente que se trata de una implementación en lenguaje `C`.

Se han mencionado en varias fuentes como en [4] las aplicaciones precisas dentro del *kernel* como podrían ser:

- Calendarizadores anticipatorios, de *deadline* y calendarizadores *CFQ* para la entrada/salida.
- Los *drivers* para CDs y DVDs.
- El código para el *timer* del sistema operativo los usa para organizar peticiones fuera de tiempo.
- El sistema de archivos `ext3` los utiliza para poder llevar un control de las entradas de los directorios.
- Las áreas de memoria virtual (*VMAs*) se llevan en un registro representado con árboles rojo-negros, así como también descriptores de archivos, llaves criptográficas y paquetes de red.

Ahora describo a detalle la importancia de los *rbtrees* en el calendarizador *CFQ I/O* del *kernel* (*Completely Fair Queue Scheduler for Input/Output* de acuerdo a [35]. La calendarización de los procesos e hilos es esencial en un sistema operativo para usar el *hardware* que está a disposición de la mejor forma posible, como son los procesares. La responsabilidad del calendarizador es la de distribuir el uso del procesador a diferentes tareas y se puede

lograr de muchas formas diferentes, como es asignando prioridades fijas o dinámicas a estas tareas y calendarizar conforme a ello. Esto es, si una tarea tiene prioridades P_1, entonces tendrá que esperara a que terminen de usar el CPU todas las otras tareas con prioridades $P_x > P_1$; esto en consecuencia da lugar a que se pueda tener *hambruna* (*starvation*), pues las tareas con mayor prioridad podrían ocasionar que las *menos* importantes no se ejecutaran por largos periodos de tiempo.

En contraparte, un calendarizador de tipo *CFQ* como el que se usa para la entrada/salida (procesos que necesitar hacer lecturas y escrituras del disco). En este calendarizador, cada núcleo del CPU tiene su propia cola de ejecución y cada tarea además tiene asociado un valor de *bondad* (*niceness* o *nice value*), así como un *peso*. El primer valor indica qué tan *bueno* es la tarea específica en relación a las otras taras, así que cuando tiene un alto valor de bondad, entonces tiene una prioridad menor pues será más factible que ceda el uso del núcleo a otros procesos (tareas).

La porción del ancho de banda de CPU que se le brinda a un proceso es directamente proporcional a su peso, i.e., $\frac{L_t}{L_{cfs_rq}}$, en donde L_t denota al peso de la tarea (*load weight*) y el denominador es la suma de todos los pesos de las tareas en la cola de ejecución. Con esto en mente, la forma en que el calendarizador determina el orden en que se asignarán las tareas para el uso del CPU es con un árbol rojo-negro —con operaciones aseguradas del orden $O(\log n)$ de búsquedas, inserciones y eliminaciones— y con el tiempo de ejecución virtual (*virtual runtime, vruntime*) de las taras. Este último concepto es el que se necesita para lograr un *fairness* en la calendarización (*justicia*), pues toma en cuenta tanto del tiempo de ejecución de la tarea como su valor de bondad.

De acuerdo al *vruntime*, una tarea con una carga alta y por ende una prioridad grande tendrá acceso más seguido al CPU que su contraparte pues la ecuación para el *vruntime* es $\frac{E}{L_t} \times NICE\_LOAD$, con E el tiempo de ejecución de la tarea y $NICE\_LOAD$ el valor por omisión de bondad. La ventaja presentada con este esquema es que los procesos no pueden completamente tener hambruna porque este *vruntime* es también proporcional al tiempo que las tareas toman en ejecutarse.

El árbol rojo-negro hace su aparición al servir para los procesos o hilos (tareas descritas antes), en donde se ordena de acuerdo al *vruntime* de ellos. De esta manera el nodo de más a la izquierda contednra aquella tarea con el **vruntime** menor y por extensión el que más requiere del uso del ancho de banda del procesador. Luego de que una tarea termina su ejecución de la porción de tiempo que le fue asignada, es reinsertada dentro del árbol rojo-negro, pero con su *vruntime* actualizado. De esta forma, el calendarizador (*scheduler*) no tiene más que tomar siempre el nodo que está más a la izquierda en el árbol.

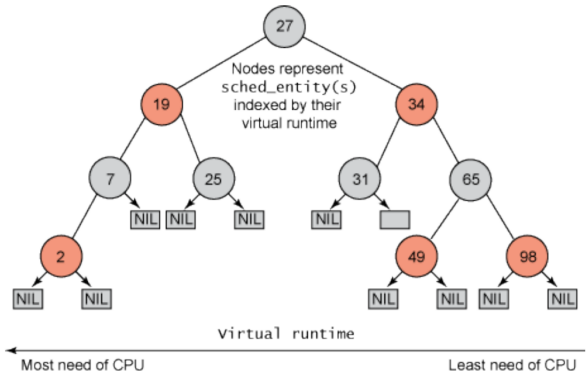


Figura 2.1: Ejemplo de un árbol rojo-negro usado en el calendarizador *CFQ* para la entrada/salida con el *vruntime* de las tareas en los nodos.

2.2. Árboles *Splay*

2.2.1. Implementación de *cachés*

La aplicación mencionada en [7] y [3] es la que puede dársele a los árboles *splay* para implementar *cachés*. Este tipo de árboles puede tener tiempos de ejecución para búsquedas más rápidos que los de árboles binarios de búsqueda aleatoriamente construidos en muchos casos. El hecho de llevar a la raíz del árbol al nodo accesado hace que se explote correctamente el principio de localidad que se asume para la implementación de *cachés*: un objeto recientemente accesado es probable que se vuelva a querer accesar en el futuro próximo.

La disparidad en el tiempo de acceso al procesador y la memoria principal sigue creciendo y los arquitectos de computadoras cada vez prestan mayor atención de la memoria *caché*, habiendo de hecho tenido que desarrollar más niveles en la jerarquía de memoria de las computadoras para poder mitigar esta diferencia en los tiempos de acceso así tener los objetos (*items*) recientes en localidades que son más rápidas de acceder. El hacer uso de estructuras que se guían también por este principio como puede ser un árbol *splay* para organizar la información puede ser útil para la implementación de memorias *caché* como se expone en los resultados de [7].

2.2.2. Compresión de datos y codificación dinámica de Huffman

En el curso de algoritmos revisamos los árboles de Huffman y los conceptos básicos en torno a la compresión de datos explicada en [15]. En efecto, los algoritmos de compresión de datos toma como entrada un cierto texto fuente y produce como salida el texto comprimido, reduciendo la cantidad de información redundante tanto como sea posible por la *entropía*. Los códigos de Huffman se encuentran dentro de las técnicas más utilizadas para la codificación sistemática de datos y ya es un método que ha sido usado por varios años. Los algoritmos de compresión de Huffman requieren de esquemas de balanceo de árboles y la similitud entre esta necesidad y lo que se logra con a operación de *splaying* desembocó en que se pensara en usarlos en este contexto.

Para ver cómo funciona, consideremos primero la representación arbórea de los códigos de prefijos: los árboles de Huffman. En este esquema, cada letra de la entrada queda representada por un código de longitud variable y se busca que las letras más comunes tengan longitudes más cortas, permitiendo por el otro lado que las menos comunes

tengan códigos más extensos. Estos códigos de prefijos —un código usado en el texto de entrada no puede ser prefijo de ningún otro código— pueden visualizarse como árboles, en donde cada hoja del árbol tiene asociada una letra del alfabeto de entrada y seguir el camino a ella desde la raíz nos da el código de prefijo al ir concatenando las etiquetas de las aristas.

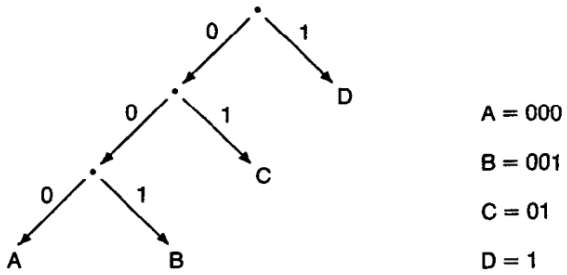


Figura 2.2: Ejemplo de un árbol de Huffman y de los códigos de prefijo.

El algoritmo convencional (el que revisamos en el curso con el Doctor Jorge Urrutia) requiere de un conocimiento previo de las frecuencias d las letras en la entrada o bien dos pasadas para que los datos puedan ser comprimidos, en donde en la primer se obtiene la frecuencia de estas. Por su parte, en los algoritmos de compresión adaptativa de Huffman solamente se necesita una pasada pues el código usado para cada letra dependenderá de la frecuencia de las letras conocidas hasta ese punto.

La aplicación de los árboles *splay* en este terreno se pensó luego de que Tarjan y Sleator[31] mostraran que los árboles *splay* introducidos en 1983 eran estáticamente óptimos[1]. Dado que los árboles de Huffman son árboles estáticamente balanceados, se pensó en usar los árboles *splay* para la compresión de datos.

El obstáculo al que se tuvo que superar como se describe en [15] fue que la operación de *splaying* aplica para cuando la información se guarda también en los nodos intermedios y no nada más en las hojas como es el caso de los árboles de Huffman. Entonces, se desarrolló una variante del *splaying* llamada el *semi-splaying* que sí sirve para los árboles de códigos de prefijos. En este esquema, el nodo que es accesado no es movido directamente a la raíz ni sus hijos son modificados, pues en su lugar el camino de la raíz al nodo se ve *recortado* en un factor de dos. Se logró demostrar que esta modificación de la operación en los árboles *splay* en el contexto de compresión de datos cumple las mismas cotas asintóticas que las de la teoría dentro de un factor constante. De esta forma, es posible generar códigos de Huffman que están sesgados hacia los símbolos que han ocurrido más recientemente y tener así una buena compresión.

[1]Esto quiere decir que si las llaves de los nodos son obtenidos de una distribución de probabilidad estática, los tiempos de acceso al árbol *splay* y los de un árbol balanceado optimizado solamente van a diferir a lo más por un factor constante cuando se amortiza este tiempo.

Capítulo 3

Estructuras de datos de gráficas planas

3.1. Lista de aristas doblemente ligadas (DCEL)

3.1.1. Sistemas de información geográfica (*GIS*)

Como se detalla en [25], los sistemas de información geográfica son paquetes de *software* que tienen la funcionalidad de almacenar datos geométricas para realizar operaciones complejas con ellos. Por ejemplo, puede interesar reportar en un cierto momento la porciones de tierra que se van a inundar cuando un cierto río alcanza un cierto nivel o bien ir analizando los costos, beneficios y riesgos asociados con el desarrollo de actividades industriales en un cierto lugar. Para lograr todo esto, gran parte se deja a las labores geométricas de localización de geométrica de datos, formas, proximidades y distribuciones espaciales y como la cantidad de datos suele ser de gran magnitud, se requieren mecanismos eficientes para el análisis y manipulación de la información; es por ello que los que desarrollan este tipo de paquetes trabajan con estructuras de datos geométricas que permiten una representación adecuada del terreno geográfico.

En clases estudiamos cómo las *Doubly-connected edge lists* sirven en un sentido general para tener más información relacional entre los vértices, regiones y aristas de una subdivisión plana y cómo aquí las aristas eran representadas cada una por dos aristas gemelas. En [9] se describe también la importancia de esta estructura para la creación de un paquete de mapas llamado `CGAL` (*Computational Geomtry Algorithms Library*), haciendo énfasis en su uso en aplicaciones geográficas. Los autores describen a detalle el diseño e implementación modular de los algoritmos para su paquete, pero en particular reclacan la separación de clases en su diseño para los *mapas planos* y *mapas topológicos* para lograr una separación útil entre la información geométrica y la topológica.

Debido a que las *DCEL* tienen la capacidad de manejar hoyos dentro de las regiones, esto tiene el efecto de que pueden ser ampliamente usadas en las *GISs*, pues en otro caso solamente se podría trabajar con representaciones geográficas que requieren que cada cara (región) del mapa estuviera simplemente conectado como se describe en [9]; por ejemplo, puede servir para representar ríos o cordilleras dentro de una región. En particular, los autores utilizan para sus mapas topológicos que tiene las funcionalidades de alto nivel una estructura de este tipo para el almacenamiento y manejo de los apuntadores de la información que como tal se almacena como se ve en su siguiente diagrama:

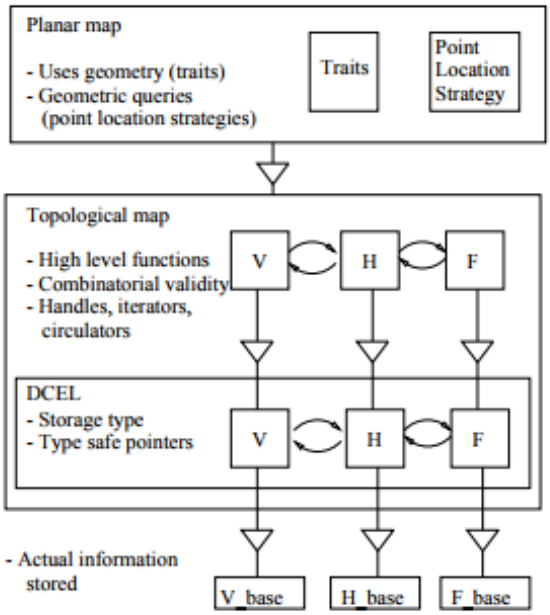


Figura 3.1: Estructura y separación llevada a cabo en [9] para la separación de la información geográfica y topológica.

Así, la capa intermedia para el mapa topológico es usada como un contenedor que almacena los objetos de la capa inferior y añade la funcionalidad requerida para manipularlos como es por ejemplo recorrer las aristas alrededor de una cara.

Más aún, como vimos en el curso, cada registro de la *DCEL* tiene la capacidad de almacenar información adicional llamada *información de atributos*, lo que podría ser usado por ejemplo en una *GIS* para indicar qué tipo de vegetación o fauna es la predominante en cada una de las regiones del mapa geográfico.

3.2. Localización de puntos (Subdivisión trapezoidal)

3.2.1. Búsquedas de intersecciones poligonales

En [10], Edelsbrunner y Maurer describen cómo por mucho tiempo había habido avances en el área de búsquedas de intersección de objetos ortogonales, esto es, siendo uno de sus ejes paralelos a uno de los ejes coordenados y se dice que dos objetos se intersecan en caso de tener mínimamente un punto en común. A partir del éxito en sus investigaciones pasadas, así como la de otros colegas en el tratamiento de intersecciones de rectángulos, los condujo a aventurarse en dar un tratamiento más general a este problema e involucrarse en la intersección de polígonos simples. Básicamente se busca dado un conjunto de polígonos simples en el plano con un número de aristas acotado y un polígono simple de consulta, determinar todos los polígonos originales que lo intersecan.

Los autores se dedican a seccionar el problema en cuestión en tres subproblemas más sencillos de resolver y que son de interés independiente:

- Dado un conjunto de polígonos simples con un número de aristas acotado, determinar todos los polígonos que contengan a un punto de consulta.
- Dado un conjunto de segmentos de línea, determinar a partir de un segmento de línea de consulta todas aquellas que son atravesadas por este último.
- Dado un conjunto de puntos en el plano y un polígono simple de consulta, devolver todos aquellos puntos que caen dentro del polígono.

Se describe en el artículo cómo es posible combinar las soluciones a estos tres subproblemas para responder las consultas principales de intersección poligonal en tiempo $O(\log n + t)$ —con un tiempo de preprocesamiento costoso—, en donde la t denota al número de polígonos que intersecan al polígono de consulta y sería el tiempo que se tarda en reportarlos a todos.

La aplicación de la subdivisión trapezoidal está implícita en el artículo pero no carece de trascendencia. Lo que hacen Edelsbrunner y Maurer en un comienzo es establecer la importancia de que es posible dados n polígonos simples en el plano con un número de aristas acotado por K, que existan estructuras de datos tal que los t polígonos que contienen un punto pueden ser localizados y reportados en tiempo $O(\log n + t)$, reduciendo el problema a una localización de puntos en el plano. Como a lo más se tienen Kn aristas para los n polígonos, la colección de aristas establece una subdivisión plana y cada región está cubierta por algunos de los polígonos originales, el fines preprocesar para cada región de la subdivisión el conjunto de polígonos que la cubren. Así, para determinar todos los polígonos que contienen a un punto de consulta, sería suficiente con determinar en qué región de la subdivisión está el punto.

Debido a que la subdisvisión consistiría de $O(n + s)$ vértices (donde s denota el número de intersecciones que ocurren entre las aristas) y además una arista puede intersecar a lo más a $K(n - 1)$ aristas, entonces s está acotada por $K^2 n(n - 1)$, lo que desemboca en el hecho de que la subdivisión plana no estará compuesta por más de $O(n + s)$ regiones, cada una de las cuales tendrá la lisa de todos los polígonos que la cubren.

Entonces, un mapa trapezoidal que represente a la subdivisión podría crearse en tiempo esperado $O(n \log n + s \log n)$ y responder consultas sobre un punto q en tiempo esperado $O(\log n + t)$ como se probó en clase y en [1]. Finalmente, esta estructura puede aumentarse para que cada región tenga la lista de polígonos que la cubren y esta información podría obtenerse en un tiempo adicional del orden $O(n + sn)$ usando $O(n + sn)$ espacio adicional.

Es con esta estructura sobre la cual se hacen las búsquedas para poder responde a varias de las consultas necesarias para resolver los subproblemas del artículo, lo que en última instancia permite responder las consultas para búsquedas de intersección poligonal. Si bien puede parecer una aplicación puramente teórica en otro problema geométrico, el hecho de poder responder consultas de intersección poligonal sirve en la práctica para por ejemplo en el diseño de videojuegos, en donde es importante determinar si dos objetos se superponen (intersecan en alguno de sus puntos al estar por ejemplo uno encima del otro total o parcialmente) para determinar correctamente cuál será la acción a realizar en dicho caso y modelar bien el aspecto físico en el motor en que se desarrolla el juego.

Capítulo 4

Estructuras de datos geométricas

4.1. Árboles Kd

4.1.1. Astronomía

Así como se describe en [5], los árboles k-dimensionales debido a su gran popularidad han encontrado un nicho de aplicaciones en el campo de la astronomía. En clase estudiamos cómo la idea básica detrás del funcionamiento de la estructura de datos geométrica es la de recursivamente ir partiendo un conjunto de puntos por hiperplanes y almacenando la información en árboles binarios. Dicho algoritmo puede ser utilizado para muchos fines astronómicos.

En la actualidad, la astronomía moderna se encuentra en un punto en el que existen catálogos gigantescos de datos de más de cien fuentes distintas y se han ido ideando nuevas maneras de poder comprimir los datos, tener un acceso directo a ellos con base en las posiciones de los cuerpos celestes e inclusive llevar a cabo consultas rápidas con base en los mismos. Como se describe en [2], la conformación de archivos astronómicos se basa en la mayoría de los casos en técnicas de mezcla de los distintos catálogos existentes con un cierto umbral en las separaciones de las mediciones permitido, en la precisión de las mediciones de cada catálogo involucrado y otras medidas. No obstante, parece ser que el **cross-identification** (*identificación cruzada*) de distintos proyectos y fuentes sigue siendo un cuello de botella de la investigación astronómica, sobre todo en términos del análisis de datos, minería de datos (*data mining*) y análisis estadístico.

Los árboles *kd* tienen una aplicación importante y reciente para el problema de la identificación de catálogos de datos astronómicos parecidos al ser un particularización del problema del vecino más cercano. Lo que se suele hacer es que para verificar que exista un cruce entre dos catálogos, un objeto de un primer catálogo A tiene que ser comparado con todos los objetos de un segundo catálogo B y evidentemente esto requiere de una operación de **join** entre tablas de bases de datos y tiene una complejidad del orden $O(n^2)$, además de que en muchos casos puede haber límites en términos de memoria. Con el uso de los árboles *kd* se desarrolló en [5] un algoritmo llamado *Hierarchical Triangular Mesh* (*HTM*) de indexado espacial con un mecanismo avanzado para poder indexar rápidamente a objetos que se localizan en esferas.

Concretamente, el algoritmo funciona primero dividiendo el cielo en pequeños triángulos todos iguales entre si y después para cada uno de ellos se construye un árbol kd para poder encontrar el vecino más cercano. El resultado indica que el *cross-matching* de dos catálogos astronómicos con hasta una decena de objetos puede llevarse a cabo en una hora.

Para ilustrar más lo que se hace en este proceso, supongamos que contamos con dos tablas A y B como las que describí antes con objetos o datos astronómicos. Ahora, pensemos que A tiene menos datos que B. Lo que se hace es transformar cada objeto a un número de índice (para ambas tablas) y se divide el cielo en triángulos iguales como dije antes. Para cada triángulo luego se insertan los objetos de la tabla B de acuerdo a los datos de estos en su árbol kd y luego lo mismo pero para los objetos de la tabla A y se encuentra el vecino más cercano o los n vecinos más cercanos en el árbol kd. Así, es posible determinar si un objeto vecino está asociado al dato de la otra tabla al analizar la distribución de las distancias de acuerdo a otras reglas estadísticas.

Ocurre que las aplicaciones astronómicas de los árboles kd van más allá de la anterior y pueden ser usadas para otras medidas como los *photometric redshifts* (estimados de la recesión en las velocidades de objetos como galaxias) y en problemas de clasificación. Tal ha sido su éxito que en [12] se detalla una biblioteca llamada `HEALPix` con propósitos astronómicos para analizar enormes cantidades de datos con ayuda de árboles kd. La búsqueda de los vecinos más cercanos en tiempo $O(\log n)$ parece ser uno de las propiedades más usadas de los árboles kd, con n el número de puntos guardados en el árbol. En este proyecto que describo, el autor se valió de la biblioteca `ANN` (*Approximate Nearest Neighbors KD Tree*) escrita en `C++` por Mount y Arya. También se permite en su implementación elegir entre diferentes métricas para llevar a cabo las consultas de vecinos más cercanos, siendo la más intuitiva la distancia euclidiana.

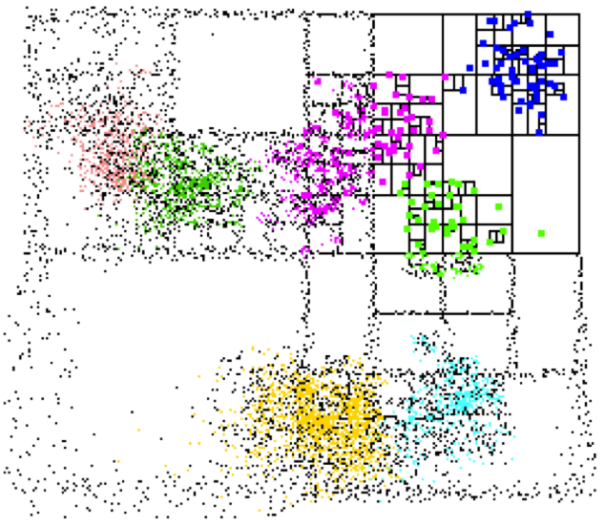


Figura 4.1: Representación bidimensional de un árbol kd utilizado dentro del proyecto de [12].

4.1.2. Gráficos y visión por computadora

También los árboles *kd* son usados en el campo de graficación y visión como se detalla en [21]. Como se explica, se usan básicamente para poder acelerar el proceso de recuperación de grandes conjuntos de entidades geométricas de $\mathbb{R}^k$. No obstante, a pesar de facilitar una tarea que de otra forma tendría que hacerse con fuerza bruta, el proceso de construcción de los árboles *kd* en este tipo de aplicaciones sigue siendo un obstáculo considerable en muchas aplicaciones y se está actualmente explorando la forma en que podría aprovecharse el paralelismo de los multiprocesadores y se han creado árboles *kd* paralelizados como los *SAH* o los *FLANN*.

Entre las aplicaciones mencionadas en el artículo en este ámbito de *rendering* fotorealístico y simulaciones de ambientes virtuales dinámicos se mencionan la del modelado tridimensional a partir de puntos (*Point based 3-d modeling*, las simulaciones basadas en partículas (*Particule based simulation*) y las búsquedas en imágenes (*Image search*). Otra aplicación importante es la del seguimiento en tiempo real de rayos (*Real time ray tracing*).

Es el caso que en muchos sistema que generan *renders* —muchos usados por arquitectos, diseñadores de interiores o de video juegos— como `OptiX` simulan efectos tales como la reflexión y refracción de la luz o la iluminación indirecta a través del seguimiento de haces de luz a través de las escenas. Los sistemas como este organizan una escena geométrica en una estructura geométrica como puede ser un árbol *kd* para poder llevar a cabo futuras consultas acerca del punto más cercano que es intersecado por un rayo de luz.

4.2. Árboles de segmentos

4.2.1. Problema de Klee

En las fuentes [11] y [8] se describe una aplicación curiosa de los árboles de segmentos en otro problema. En el año de 1977, el matemático norteamericano especializado en conjuntos convexos y optimización combinatoria Vistor Klee se hizo la pregunta de qué tan rápido sería posible calcular la longitud de la unión de una colección de intervalos en la recta real. Lo primero en que se fijó es que se podría hacer en tiempo $O(n \log n)$ con un algoritmo de ordenamiento. Poco tiempo después, Jon Bentely comenzó a interesarse en la generalización de este problema en el caso bidimensional: *dada una colección de cajas rectangulares alineadas a los ejes, qué tan rápido era posible calcular la unión de sus áreas*.

En términos muy generales, la idea para resolver este problema en el caso de dos dimensiones en tiempo óptimo $O(n \log n)$ usa una versión dinámica de la solución unidimensional, con un barrido de línea que va de izquierda a derecha a lo largo de los rectángulos. El punto es que en todo momento se mantiene la intersección de los rectángulos y la línea de barrido —que es una colección de intervalos— en un árbol de segmentos:

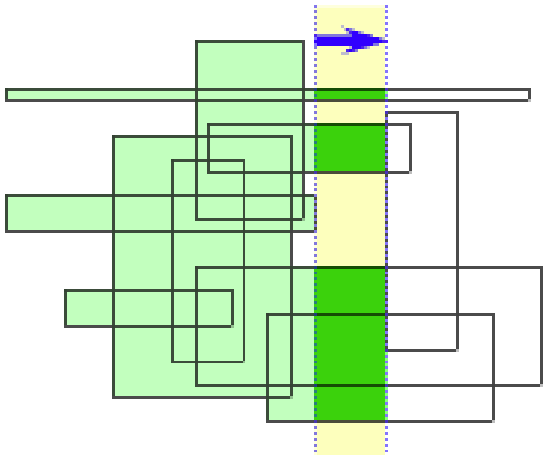


Figura 4.2: Ilustrando el barrido de línea en el algoritmo de Bentley.

Así, cada vez que la línea de barrido alcanza una de las líneas verticales de un rectángulo, el árbol de segmentos debe ser actualizado junto con la totalidad del área barrida. Un nuevo intervalo puede ser añadido, uno existente puede ser eliminado o bien se pueden tener operaciones de unión y separación. Al usar el árbol de segmentos, la actualización ocurre en tiempo $O(\log n)$ y como cada rectángulo tiene solamente dos aristas verticales, entonces el número de actualizaciones es del orden $O(n)$, haciendo que el algoritmo tenga un tiempo de ejecución en su totalidad del orden $(n \log n)$, siendo así óptimo para el caso bidimensional.

En caso de querer generalizar más el problema a más dimensiones, la estrategia de Bentley puede manejar el problema d-dimensional al dividirlo en problemas $n(d-1)-$dimensionales. No obstante, en el año de 1991, Mark Overmars y Chee Yap usaron ya no los árboles de segmentos, usando una estructura parecida a los árboles-kd que ahora guardaban cajas bidimensionales en lugar de intervalos y en donde entonces las inserciones y eliminaciones podían hacerse en tiempo $O(\sqrt{n} \log n)$. Por tanto, se logró así ahora para el caso d-dimensional un tiempo de ejecución del orden $O(n^{\frac{3}{2}} \log n)$ y el problema sigue siendo sin embargo un área abierta de investigación, pues se sabe que la cota inferior es $\Omega(n \log n)$ y como se indica en [11], conforme se incrementa el número de dimensiones, el problema de vuelve NP-Difícil.

En [11] se estudia una versión discreta y dinámica del problema de Klee en donde el volumen de una caja es definido como la cardinalidad de su intersección con un conjunto de puntos finito $\mathcal{P}$ y tanto las cajas como los puntos son sometidos a inserciones y eliminaciones. Como se indica, el problema de Klee es importante pues tiene aplicaciones inherentes en aplicaciones que tienen que ver con datos multi-atributos. Se utiliza en el caso dinámico un árbol de segmentos *multinivel* en donde representan n cajas en el espacio d-dimensional. Esta estructura requiere espacio del orden $O(n \log^{d-1} n)$ y tiempo para su contrucción $O(n \log^{d} n)$ además de que las consultas sobre un punto las contesta en tiempo $O(\log^{d-1} n)$.

Capítulo 5

Heaps

5.1. Min-max heaps

Como revisamos en el curso, el punto fuerte de esta estructura de datos es justamente lo que la caracteriza: al ir cambiando el nivel en la estructura arbórea asociada entre un nivel *min* y uno *max*, entonces los *min-max* heaps permiten obtener en tiempo constante el menor y el mayor elemento, así como eliminaciones del orden $O(\log n)$. Esto hace que tengan un uso importante en la implementación de *DEPQ* (*Double-ended priority queues*). A continuación describo algunas de las posibles aplicaciones:

5.1.1. Ordenamiento externo

El *external sorting* es una aplicación las colas de prioridad de doble extremo que pueden implementarse con *min-max* heaps como se describe en [28].Al querer usar este ordenamiento es porque se tienen más elementos de los que se pueden almacenar en la memoria principal de una computadora. Entonces, los elementos a ser ordenados están inicialmente en un disco y la secuencia ordenada deberá dejarse en dicho disco. La estrategia de ordenamiento usa la cola de prioridades de doble extremo de la siguiente forma:

1. Se leen tantos elementos como sea posible y se colocan en esta cola de prioridades, que puede ser en nuestro caso el *min-max heap*.
2. Eventualmente los elementos dentro de la DEPQ van a tener al elemento pivote de la mitad de los elementos.
3. Se procede a leer el resto de los elementos, pero se hace una comprobación en cada caso:
   - En caso de que el elemento leído no exceda al menor elemento de la DEQP (que con el *min-max heap* se obtiene en $O(1)$), entonces se obtiene el siguiente elemento como parte del grupo izquierdo.
   - En caso de que sea mayor o igual al elemento mayor de la DEPQ (de nueva cuenta al usar un *min-max heap* es en tiempo constante), se coloca como parte del grupo derecho.
   - Finalmente en cualquier otro caso se tiene que quitar al menor o al mayor elemento de la DEQP (en tiempo constante) de forma aleatoria. Si se quita al mayor elemento, se coloca como parte del grupo derecho; en caso de tomar la otra decisión, el elemento extraído será parte del grupo izquierdo.

4. Se obtienen los elementos de la DEPQ en su orden como el grupo central; ya quedaron ordenados.

5. Se ordena ahora los grupos de la izquierda y de la derecha de forma recursiva.

5.1.2. Control de ancho de banda en la red

McKenzie [23] describe a grandes rasgos como los *heaps* tradicionales pueden ser usados para el control del ancho de banda en cualquier red. En este caso, un enrutador (*router*) se encuentra conectado a una línea con un ancho de banda limitado y en caso de que no haya suficiente, entonces el *router* deberá mantener una cola de prioridades que puede ser implementada con un montículo tradicional para que los datos más importantes sean transferidos antes de los menos importantes cuando el ancho de banda esté disponible. En caso de que el ancho de banda fuera muy limitado, podría implementarse una cola de prioridades de doble extremo con un *min-max heap* para que pudiéramos de igual forma detectar en tiempo constante los datos menos valiosos y así proceder a estos enviarlos por otra ruta que potencialmente podría ser más lente, con más saltos y mayor congestión, pero asegurándose que los de mayor prioridad serán los únicos que estarán viajando por la ruta principal que tiene el ancho de banda a veces limitado.

5.1.3. Árbol de búsqueda *min-max in-place*

Minati y sus colegas describen en [24] una aplicación de los *min-max heaps* al hibridizarlos con los árboles binarios de búsqueda clásicos para guardar puntos en $\mathbb{R}^2$ *in-place*, esto es, pudiendo guardarlos dentro de un arreglo deforma tal que cada entrada solamente guarde un punto del conjunto de puntos y ningún punto es guardado en más de un lugar del arreglo.

Ellos muestran a detalle cómo es posible responder a todas las consultas que se podían en las estructuras por separado en la nueva estructura mixta con el mismo tiempo de ejecución y usando espacio adicional constante $O(1)$. Además, esta estructura que se logra de la aplicación de los conceptos detrás del *min-max heap* encuentra su uso en la práctica para lograr la implementación de un algoritmo *in-place* que sirve para enumerar todos los rectángulos maximales vacíos con ejes paralelos a los ejes coordenados en una región rectangular $R \subset \mathbb{R}^2$ en tiempo $O(m \log n)$ en donde m se refiere a la cantidad de rectángulos vacíos maximales.

5.2. Heaps binomiales

5.2.1. Algoritmo para árbol generador de peso mínimo de Cheriton-Tarjan

El algoritmo para encontrar árboles generadores de peso mínimo (*MSTs, Minimum Spanning Trees*) de Cheriton-Tarjan fue diseñado para gráficas arbitrarias y tiene un tiempo de ejecución del orden $O(m \log \log n)$ con $|E| = m$ y $|V| = n$. Creado en 1976m usa como estructura de datos un *leftist heap* para ir guardando las aristas, aprovechando el hecho de que estos montículos pueden ser mezclados en tiempo logarítmico, además de que cada *heap* es binario y así cada nodo tiene a lo más dos descendientes.

Dos años después, en 1978 se crearon los *heaps* binomiales como una estructura de datos más *limpia* y como vimos en clase permite llevar a cabo la mezcla en tiempo $O(\log n)$. Desde ese año la mayoría de los algoritmos que requieres de *heaps* que se puedan mezclar tienden a usar *heaps* binomiales o *heaps* de Fibonacci. El algoritmo de

Cheriton-Tarjan ha sido adpatado como se indica en [17] al hecho de que los *heaps* binomiales no usan árboles binarios y en consecuencia la optimización principal del algoritmo que e la de eliminar aristas innecesarias haciendo un recorrido DFS sobre el *heap* tiene que ser adaptada a los *heaps* binomiales.

La idea concreta que se exhibe en [17] es la de un algoritmo glotón (*greedy*) en donde se comienza a partir de un bosque en donde cada árbol es un solo nodo (*singleton*). Posteriormente, se tiene que ir elgiendo en cada paso un árbol T del bosque F y elegir la arista de menor costo e que conecta a un vértice de T con un vértice de T', agregabndo la arista al bosque —que sería mezclar T y T' en un solo árbol—. Para lograr la implementación, Cheriton y Tarjan se basaron en el algoritmo de *Kruskal* y determinaron que habría que mantener a los árboles en un cola, para luego:

1. Elegir el primer árbol de la cola, T.
2. Elegir la arista menos pesada r que conecta con otro árbol T'.
3. Quitar a T' de la cola.
4. Conectar T con T' a través de la arista e.
5. Agregar el árbol resultante de la unión (mezcla) en la cola.

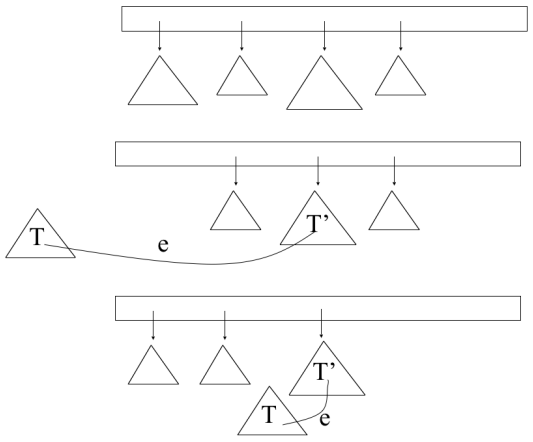


Figura 5.1: Ilustrando el proceso de mezcla en la cola, implementada con un *heap* binomial.

La idea de la modificación que usa *heaps* binomiales es:

- Que los vértices de cada árbol T estén en un conjunto que funcione como una estructura de datos de tipo *Union-Find*.
- Las aristas con uno de sus extremos en T son guardadas en un *heap*, usando uno de tipo binomial y se denotaría como $h(T)$. Así, es posible encontrar rápidamente la menor arista (de menos peso o costo) de las guardadas en el montículo, luego poder hacer una búsqueda del otro extremo que no es el que conecta con el vértice de T en el otro árbol T' y proceder a hacer la unión de los dos árboles para agregarlos a la estructura haciendo la mezcla de los *heaps* de sus aristas.

Capítulo 6

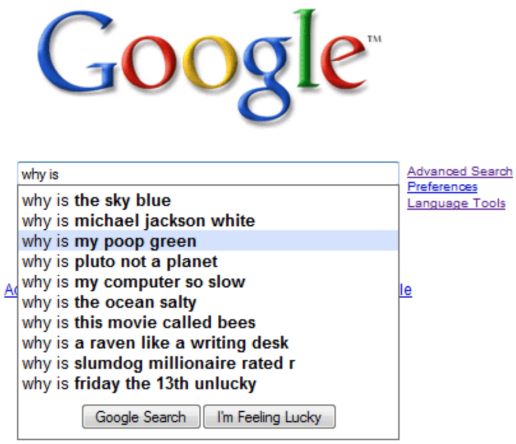
Estructuras de datos para cadenas

6.1. *Tries*

Los árboles *trie* (re**trie**val, *recuperación*) tienen aplicaciones clásicas como las que se se describen a detalle en para el autocompletado (*autocomplete*) y la *autocorreción* (*autocorrect*). Veamos a ambas:

6.1.1. Autocompletado

El autocompletado es una funcionalidad hoy en día presente en muchos lados, desde los navegadores web, como aplicaciones y hasta en los sistemas de archivos cuando se sugieren comandos al presionar `TAB` luego de comenzar a escribir el prefijo de alguno o en los editores de texto como `Sublime Text`, `Emacs` o `Vim` que pueden completar nombres de funciones, rutinas, variables, etcétera.



Los árboles *trie* sirven para poder implementar estos sistemas de autocompletado [26] y lo hacen de una manera muy directa. En primer lugar se tienen que seguir los apuntadores de la cadena que el usuario lleva escrita, la

cual representa un prefijo de una palabra en el buscados o de un comando, por ejemplo. Posteriormente, basta con enumerar todas las cadenas que son guardadas en los nodos de dicho subárbol.

Así, básicamente se hacen dos pasos:

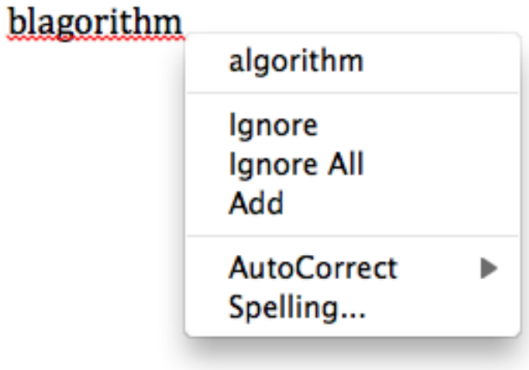
1. Un `find()` al último nodo en la cadena del prefijo insertado por el usuario. Tiene una complejidad en el peor caso del orden $O(m)$ donde m es la longitud de la cadena más larga almacenada en el *trie*.
2. Ejecutar un `DFS` usando el nodo previamente localizado como raíz para poder crear una lista de sugerencias o irlas dando a la par que se va recorriendo.

Por ende, este procedimiento tendría una complejidad del orden $O(nm)$ en donde n es el número de palabras y m la longitud de la cadena más larga almacenada.

6.1.2. Autocorrección

Ahora en muchos editores de texto y clientes de correo electrónico se tiene incorporada la funcionalidad de que al escribir una cadena no reconocida, se sugieran varias alternativas que se asemejan en cierto grado y que probablemente quisimos escribir en primer lugar; esto es fundamental para la corrección de errores ortográficos y es el corrección del *spell-checking* moderno. Las cadenas reconocidas evidentemente deben estar almacenadas en alguna estructura de datos como diccionarios o árboles, y cuando se escribe una se procede a buscarla dentro de ésta para ver si se encuentra o no, significando entonces que hubo o no un error ortográfico.

Luego de esta primera fase, se tiene que proponer la lista de potenciales palabras que pudieron haberse querido escribir en su lugar y ordenarlas de forma tal que la más probable aparezca encima de las demás; esta funcionalidad puede ser fácilmente implementada con ayuda de un árbol *trie*, brindando las diferentes sugerencias como se explicó antes en el caso del autocompletado.



Para poder proponer las palabras que quizás se quisieron escribir en lugar de la no reconocida se describe el concreto de la *distancia de edición* (*edit distance* ([16]), que viene a ser el número de inserciones, eliminaciones o sustituciones que se necesita para transformar una cadena en otra. También se le conoce como la distancia *Levenshtein* y calcular la distancia de una palabra a todas las demás almacenadas en una estructura podría ser muy costoso

e impráctico. Es por este motivo que se establece desde un principio una distancia máxima hacia las palabras de las cuales se harán sugerencias, se calcula la distancia de edición desde la raíz del árbol y recursivamente desde los hijos, deteniéndose cuando la distancia de edición rebase la que se dictaminó en un inicio para así solamente recorrer los nodos *razonables* del árbol para irlos sugiriendo.

6.1.3. Criminología

También se suele usar esta estructura de datos [27] en el área de la criminología. Por ejemplo, pueden servir en una escena de un crimen si los atacantes escapan en un vehículo y los testigos solamente logran recordar un prefijo de la placa. Con este árbol se podrían explorar los subárboles para encontrar más similitudes con las del vehículo cuyas placas empezaban con el prefijo recordado. No obstante, una aplicación interesante es la que se les ha dado en las tablas de ruteo.

6.1.4. Tablas de ruteo

Los árboles *trie* son ampliamente usados en el algoritmo del prefijo común más largo usado dentro del protocolo *IP* (*Internet Protocol*) —en particular en las tablas de ruteo— en redes de computadoras. Este algoritmo es el que usan los enrutadores (*routers*), los equipos que determinan por cuáles caminos deberán de transferirse los paquetes en la red para que puedan llegar a sus destinos, eligiendo una entrada dentro de los registros de la tabla de ruteo del *router*. Cada entrada en la tabla de ruteo puede corresponder a una red y en consecuencia una dirección de destino para un mensaje podría coincidir con más de una entrada; sin embargo, el *router* debe elegir aquella que sea la más específica (que tenga la máscara de red más alta).

En [33] se explica que hay varias estructuras de datos y algoritmos para trabajar con las tablas de ruteo. La más sencilla y fácil de implementar no es la más óptima en cuanto a tiempo de búsqueda. Es de orden lineal $O(n)$ con n el número de registros en la tabla de ruteo y consiste en representarla con una estructura linea como una lista o un arreglo; un acercamiento muy parecido es el que por omisión viene implementado en el *hardware* de los enrutadores.

Una optimización se presenta en los *routers* de *CISCO* y su tecnología **fast switching, CFS**. Aquí se emplean árboles *trie* de bits en donde las cadenas almacenadas serían las dirección IPs y con consideradas como llaves; en lugar de que las llaves sean almacenadas en los nodos internos, las llaves se determinan por sí mismas a partir de la sucesión de apuntadores de bits. De esta forma, dada una dirección de destino, se utiliza su valor (en binario) para ir recorriendo el árbol *trie* asociado a la tabla de ruteo y poder recorrer lo más posible hacia abajo, lo cual dará como resultado al unir todos los ceros y unos del camino recorrido la dirección a la cual hay que retransmitir el paquete.

La complejidad en este caso es del orden $O(n)$ con n la longitud de la llave (dirección IP destino), que en el peor de los casos para la versión 4 de IP sería de treinta y dos bits y en IPv6 de ciento veintiocho.

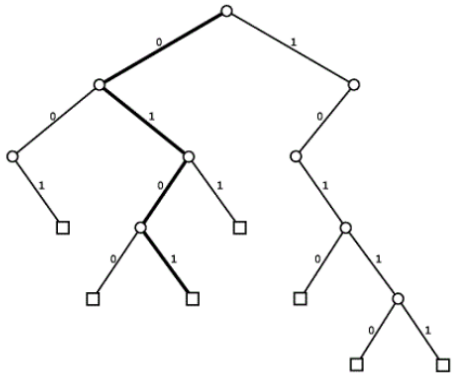


Figura 6.1: Árbol *trie* de bits para la tabla de ruteo; buscando el predijo **0101**.

6.2. Árboles de sufijos

Desde su concepción en el año de 1973 [20] y su posterior perfeccionamiento en el algoritmo de construcción de tiempo lineal por Ukkonen e 1995 para alfabetos de tamaño constante, esta estructura de datos para manipulación de cadenas ha probado contar con un sinfín de aplicaciones en distintos ámbitos como por ejemplo la compresión de datos mencionada a detalle en [22] y [20]. A continuación describo dos de las más importantes, en un contexto más específico.

6.2.1. Contaminación del *ADN*

Se trata de una aplicación interesante descrita en [34]. En esta fuente también se hace énfasis en que si bien la motivación por parte de Weiner para la construcción de los árboles de sufijos no fue en un inicio para el área de bioinformática y biología molecular, es el caso que su uso para resolver diversos problemas de combinatoria en torno a cadenas cazan perfectamente con muchos aspectos de interés que surgen al momento de querer investigar secuencias de *ADN* o *ARN* como puede ser el buscar la subcadena común más larga entre dos secuencias o encontrar apareamientos inexactos de una muestra en una secuencia larga. Tal es el caso que muchos paquetes dirigidos para esta rama de la medicina e investigación que se basan en estas estructuras de datos para cadenas han nacido en los último años como son `Multiple Genome Aligner` y `MUMmer`.

Concretamente, una aplicación clásica que se sigue mejorando es la mencionada en [6], en donde se describe a grandes rasgos el equipo bioinformático necesario para analizar y comprar grandes conjuntos de secuencias de *ADN*, tarea que se viene haciendo ya desde hace muchos años con árboles de sufijos. Sin embargo, el paradigma ofrecido por el artículo es el de incorporar tanto la idea de no tener los discos en memoria principal (*RAM*), sino en medios de almacenamiento secundario y contar con una interfaz gráfica de usuario (escrita en `Java` y usando módulos de `C`) que ayude a los laboratoristas. La idea detrás de este *STS* (*Suffix Tree Searcher*) es la de proporcionar una herramienta fácil de usar para poder indexar, comparar y estudiar grandes bases de datos de secuencias de *ADN*, en donde cada una puede llegar a decenas de miles de millones de nucleótidos.

Una aplicación un tanto menos conocida es la de la solución al problema de contaminación del *ADN*. Ocurre que cuando se procesa *ADN* en un laboratorio no es fácil evitar que secuencias de *ADN* del ambiente y foráneas contaminen las secuencias de interés que se piensan estudiar de un cierto organismo. El problema radica en que inclusive la más mínima cantidad de contaminación puede propiciar el desencadenamiento de *PCR* (*Polymerase Chain Reaction*) que provoque el hecho de que parte de la secuencia contaminante se inserte en la secuencia objetivo, pudiendo provocar que todo el experimento sea en vano.

No obstante, usualmente se conocen los potenciales contaminantes de secuencias de *ADN* en específico como pueden ser vectores de clonación. Al problema puede dársele la vuelta al tomar esto en cuenta y combinarlo con el uso de árboles de sufijos. Ahora, al conocer la secuencia de *ADN* de interés —llamada la cadena S y un conjunto de cadenas C —las de los posibles contaminantes—, el punto ahora es el de encontrar todas las subcadenas de todos los $T \in C$ que están presentes en S y no al menos tan largas como una longitud fija l; estas serían las subcadenas que probablemente pudieron haber contaminado la secuencia de *ADN* objetivo S.

Con un árbol de sufijos generalizado de las cadenas S y todas las de C se procede a marcar a todos los nodos internos v que son parte de un camino que representa un sufijo de S y también de otro camino que representa un sufijo de alguna cadena del conjunto C. Finalmente, de entre los nodos marcados anteriormente se toman aquellos que están a una distancia en el camino de al menos l serían así estas partes las susceptibles de haber sido contaminadas en la secuencia.

Bibliografía

[1] Mark Overmars Mark de Berg Marc van Kreveld. *Computational Geometry: Algorithms and Applications*. Springer, 1997.

[2] Nieto-Santisteban M. Budavári T. Szalay A. S. *Probabilistic Cross-Identification of Astronomical Sources*. Harvard University, 2008.

[3] Ravi Teja Cheruku. *Splay Trees*. Indiana State University, 2011.

[4] Corbet. *Trees II: red-black trees*. lwn.NET, 2006.

[5] Yongheng Zhao Dan Gao Yanxia Zhang. *The Application of kd-tree in Astronomy*. Cornell University, 2008.

[6] Song-Han Lin David Minkley Michael J. Whitnet. *Suffix tree searcher: exploration of common substrings in large DNA sequence sets*. US National Library of Medicine National Institutes of Health, 2014.

[7] Charles U. Martel Eric K. Lee. *When to use splay trees*. Department of Computer Science, University of California at Davis, 2006.

[8] Jeff Erickson. *Klee's Measure Problem*. University of Illinois at Urbana-Champaign, 2011.

[9] Iddo Hanniel Eyal Flato Dan Halperin. *The Design and Implementation of Planar Maps in CGAL*. Tel Aviv University, Israel, 1988.

[10] H.A. Maurer H. Edelsbrunner. *Polygonal Intersection Searching*. Technical University of Graz, University of British Columbia, 1988.

[11] Subhash Suri Hakan Yildiz John Hershberger. *A Discrete and Dynamic Version of Klee's Measure Problem*. Canadian Conference on Computational Geometry, 2011.

[12] Matthew J. Holman. *KD Trees in the Sky with MPI*. Harvard University, 2013.

[13] Udi Wieder James Aspnes. *The expansion and mixing time of skip graphs with applications*. Springer-Verlag, 2005.

[14] Gang Zhou Jianjun Yu Hao Su. *SNet: Skip Graph based Semantic Web Services Discovery*. ACM, Corea, 2007.

[15] Douglas W. Jones.

[16] Seny Kamara. *Autocorrect and Autocompleta: CS16: Introduction to Data Structures and Algorithms*. Brown University, 2013.

[17] Haim Kaplan. *Advanced topics in data structures: Binomial heaps, Fibonacci heaps,and applications*. Tel Aviv University, 2007.

[18] Philip Koopman. *Stack Computers: the new wave. Why are Stacks used in Computers?* First Edition. Carnegie Mellon University, 1989.

[19] Gerald W. Kruse. *Stack Applications*. First Edition. Juniata College, Pennsylvania, 2017.

[20] N.Jesper Larsson. *Extended Application of Suffix Trees to Data Compression*. 1999.

[21] Victor Lu. *Multicore Construction of KD Trees with Applications in Graphics and Vision*. University of Illinois at Urbana-Champaign, 2013.

[22] Moritz Maaß. *Suffix Trees and their Applications*. Stanford University, 1999.

[23] John McKenzie. *CSE 373 Data Structures and Algorithms: Lecture 13: Priority Queues (Heaps)*. Paul G. Allen School of Computer Science, 2003.

[24] Subhas C.Nandy Minati De AnilMaheshwarib. *An in-place min–max priority search tree*. Elsevier, 2013.

[25] Rene Willibrordus van Oostrum. *Geometric Algorithms for Geographic Information Systems*. College voor Promoties, Holanda, 1999.

[26] Kevin Wayne Robert Sedgewick. *Algorithms and Data Structures Fall 2007: Tries*. Princeton University, 2007.

[27] Sartaj Sahni. *Data Structures, Algorithms, and Applications in C++ Tries*. University of Florida, 2004.

[28] Sartaj Sahni. *Data Structures, Algorithms, and Applications in Java Double-Ended Priority Queues*. University of Florida, 1999.

[29] Madhu Kumar Shabeera T P Priya Chandran. *Authenticated and Persistent Skip Graph: A Data Structure for Cloud Based Data-Centric Applications*. ACM, 2012.

[30] Amritpal Singh Shalini Batra. *A short survey of Advantages and Applications of Skip Graphs*. First Edition. International Journal of Soft Computing y Engineering (IJSCE), 2013.

[31] D.D. Sleator Tarjan R.E. *Self-adjusting binary search trees.* ACM, 1985.

[32] Christian Scheideler Thomas Clouser Mikhail Nesterenko. *A self-stabilizing deterministic skip list and skip graph*. Elsevier, 2012.

[33] I. L. Kaftannikov V. V. Kashansky. *Application of TRIE data structure and corresponding associative algorithms for process optimization in GRID environment*. South Ural State University, 2016.

[34] Bálint Márk Vásárhelyi. *Suffix Trees and Their Applications*. Eötvös Loránd University Faculty of Science, 2013.

[35] Ludwig Hellgren Winblad. *CFS (Completely Fair Scheduler) in the Linux kernel*. Lund University, 2001.