1. Orden

Dada la siguiente versión del algoritmo **Selection Sort**:

```
input: Secuencia de datos en Arreglo A[]
output: Nueva secuencia B[] ordenada
selectionSort(A[]):
    B ← vacia
    for i = 0 to n - 2:
        min ← 0
        for j = 1 to n - 1:
            if A[j] < A[min]:
                min ← j
        B[i] ← A[min]
        A[min] ← +infinito
    return B
```

a) (2 puntos) Escriba el pseudocódigo para **Tournament Sort**, el cual es una mejora de **Selection Sort** al utilizar una cola priorizada para seleccionar el siguiente elemento en orden. Puede asumir que dispone de las funciones de una cola priorizada vistas en clases.

Hay varias opciones de solución, puede ser construyendo un heap de forma incremental o como se muestra a continuación:

```
input: Secuencia de datos en Arreglo A[]
output: Nueva secuencia B[] ordenada
selectionSort(A[]):
    B ← vacia
    A ← BuidHeap(A) // minHeap
    for i = 0 to n - 1:
        min ← extract(A)
        B[i] ← min
    return B
```

b) (2 puntos) Determine la complejidad de tiempo de **Tournament Sort**: ¿hay un mejor o peor caso? ¿Cuál es su complejidad de memoria?

BuidHeap es O(n). El for ejecuta n veces extract() es O(n log n), Luego Tournament Sort es O(n log n) en tiempo. Al igual que selection sort, TS no aprovecha la composición inicial del input al seleccionar el menor valor, y tampoco las funciones del heap, luego no hay un mejor o peor caso. La complejidad de memoria es O(n) ya que requiere la memoria adicional del arreglo B.

c) (1 punto) Argumente (no demuestre formalmente) que **Tournament Sort** es correcto.

El arreglo A es finito, BuildHeap() es correcto (termina), el for es sobre un numero finito de elementos y extract() es correcto (termina) luego, TS termina.
De igual forma, sabemos que extract() extrae del minHeap el menor elemento en cada paso, luego en un paso n, los n elementos extraidos son menores que los que permanecen en el heap, dado que sabemos que extract es correcto. Asi en un paso n+1, el valor extraido es mayor igual que los anteriores, por lo que TS ordena correctamente.

d) (1 punto) ¿Es **Tournament Sort** estable? Argumente su respuesta.

No lo es, básicamente porque las operaciones del heap no garantizan mantener el orden de llegada de los elementos del arreglo al realizar el BuidlHeap().

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructuras de Datos y Algoritmos 2'2024

Examen

11 de diciembre de 2024

Condiciones de entrega: Para los que rindieron la I1 y la I2, deben entregar solo 3 de las siguientes 4 preguntas. Para los que tienen Interrogación recuperativa, deben entregar entre las 3 preguntas la correspondiente a la materia de la Interrogación faltante.

Tiempo: 2 horas

Entrega: Al final de la prueba tienen 10 minutos para subir la imagen de la prueba a Canvas,**cada pregunta y el torpedo** por separado en vertical

Evaluación: Cada pregunta tiene 6 puntos (+1 punto base). La nota es el promedio de las 3 preguntas entregadas. La nota de la I recuperativa es el promedio de las preguntas recuperativas y la pregunta correspondiente del examen.

1. B+ Hash

a) (3 pts) Escriba los algoritmos usando la mejor estructura para hacerlo.

- Rango(i, f), con i,f llaves. La mejor estructura es B+ ya que la búsqueda del elemento k ¿=i es $O(log_2 d(n))$ y luego recorrer la lista ligada de hojas hasta K¡=j El algoritmo es:
  Buscar k¿=i en el árbol -¿hoja que contiene i while k¡j do If se terminó la hoja siga con hoja.next[0] inserte k en resultado k¡-k+1 done
- Record(i) La mejor estructura en Tabla de Hash con una función que distribuya uniforme O(1) El algoritmo es:
  k=h(i) if b[k] null return false else if b[k] ¡¿i recorra colisiones(k,i) else return b[k]

b) (3 pts) Escriba los algoritmos

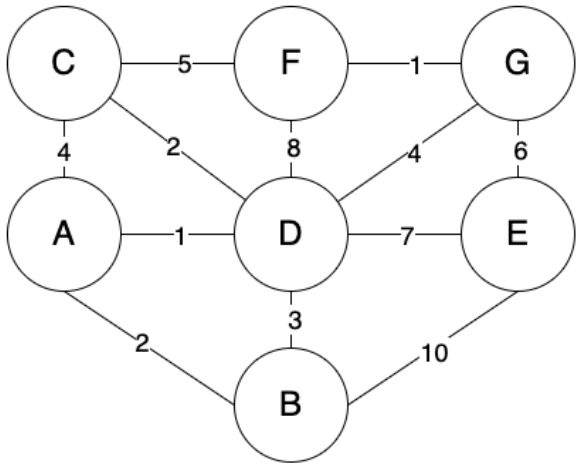
- Insert(k, r), k llave r registro. Como estamos un nuevo registro, se deben actualizar ambos índices El algoritmo es:

```
Insert(k,r)
    InsertA(k,r)
    InsertB)k,r)

InsertA(k,r)
    P=record(k, A). ------ P hoja
    If  |P| < 2d , insertA(k,r,P)
    else insertA(k,r,P); split(P +k) --[P +k]= k1,k2,.....,k2d+1

split(P+k)
    create(P1, P2,P) crea nodos P1 y P2 y copia las llaves de k1,...,kd en P1 y kd+1,.
    P.previous.next=P1;P1.next=P2;P2.previous=P1;P2.next=P.next
    Insert(Kd+1,P.father)
```

```
    P.father.p[p1,p2,kd+1] arregla los punteron en el padre de P

InsertB(k,r)
    i=h(k)
    if B[i]==null B[i]=r
    else insertcolision(k,r) --- no es necesario hacer el pseudo código


DeleteA(k)
 P=record(k, A)
 If |P| >= d+1  borrar k del nodo
 else borrar k en P y rebalancear usando unsplit o merge

 DeletaB(k)
    i=h(k)
    if B[i]==k
        B[i]=NULL o Borrado
    else delete colisión
```

3. Kruskal y Prim

Considera el siguiente grafo no direccional con costos.



Vértice	T	Distancia	Padre
A	-	∞	–
B	-	∞	–
C	-	∞	–
D	-	∞	–
E	-	∞	–
F	-	∞	–
G	-	∞	–

a) (2 puntos) Ejecuta paso a paso el algoritmo de **Prim** para determinar un árbol de cobertura de costo mínimo (**MST**), tomando A como vértice de partida. En cada paso muestra la versión actualizada de la tabla a la derecha, en que $\boldsymbol{T}$ indica si el vértice ya está en la solución.

Seguir la ejecución paso a paso desde A

b) (2 puntos) Demuestra con un contraejemplo que el árbol producido por **Prim** no necesariamente tiene las rutas más cortas desde el nodo inicial al resto de los nodos.

Podemos ver que el MST va por A-D-G-E con costo 11, pero la ruta más corta de A a E es A-D-E y su costo es 8.

c) (2 puntos) Una alternativa para determinar un **MST** es el algoritmo de **Kruskal**. Si todos los costos de las aristas son números enteros en el rango 1 a $|V|$, ¿qué tan rápido, en notación $O(\cdot)$, se puede hacer que ejecute el algoritmo de Kruskal? **Indicación:** Considera que el algoritmo incluye una inicialización, una ordenación, y finalmente la ejecución del algoritmo propiamente tal.

Kruskal toma O(V) para inicialización, O(E log E) para ordenar las aristas, y O(E (V)) para las operaciones de conjuntos disjuntos (la ejecución del algoritmo propiamente tal); por lo tanto, Kruskal es O(E log E). Ahora, bajo el supuesto de arriba y usando countingSort, podemos ordenar las aristas en O(V + E) = O(E) —ya que V = O(E). De modo que ahora Kruskal es O(E (V)).

Bellman-Ford:

a) Para que la afirmación sea verdadera, las aristas (u, v) deben ser actualizadas en el orden topológico de los vértices u, es decir, de los vértices de partida de cada arista (nota: no hay orden topológico de aristas). Si hacemos esto, entonces todas las aristas que terminen en u van a haber sido actualizadas antes de actualizar la arista (u, v) propiamente tal; y, como sabemos, u.dist depende de las aristas que terminan en u (no de las que parten de u).

b) El algoritmo consiste simplemente en ir actualizando las aristas (u, v), una vez cada una, en el orden topológico de los vértices u (es decir, de los vértices de partida): primero actualizamos todas las aristas que tienen como vértice de partida al vértice s; luego, actualizamos todas las aristas que tienen como vértice de partida al siguiente vértice a s en el orden topológico; y así sucesivamente. (Por supuesto, el primer paso, antes de actualizar cualquier cosa, es ordenar topológicamente el grafo.)

c) El algoritmo anterior es correcto por a); y claramente es O(V+E): la ordenación topológica (el primer paso) lo es, y luego las aristas se actualizan una vez cada una (recordemos que actualizar una arista es O(1)).