Nombre: Tomás Maturana

RUT: 19.186.973-7

C2 DAA

P1:

Ya que se debe acceder al k-ésimo elemento en tiempo constante en el peor caso y además el arreglo debe ocupar espacio O(n), la estructura tendrá dos arreglos A y B, uno para las inserciones al final y otro para las inserciones al principio. De este modo, si se inserta al final, se inserta en A y si se inserta al principio, se inserta en B. Además se tendrá una variable Alen, que representa el largo de A y Blen, que representa el largo de B. Además, el largo de A más el largo de B será n.

El primer elemento insertado, será insertado en A. Luego, en cada inserción al principio, se insertará el elemento en B y sumará 1 a Blen; y en cada inserción al final, se insertará el elemento en A y se sumará uno a Alen. Por esto, las inserciones serán de costo constante.

Luego, para acceder al k-ésimo elemento, se debe restar Blen a k, resultando z. Si z es positivo, se entrega A[z-1] ; si z es negativo, se busca B[|z|]. De esta manera, tambien tenemos el acceso al k-ésimo elemento en tiempo constante.

Ej: Sea A y B los arreglos de la estructura y L[] el arreglo que se está formando. Además (insertarComienzo(x) := iC x) y (insertarFinal(x) := iF x) y (acceder(k)) para acceder al k-ésimo elemento:

iC 1	-> L={1}	:”virtualmente” -> A={1}, B={}
iF 3	-> L={1, 3}	: ”virtualmente” -> A={1, 3}, B={}
iC 5	-> L={5, 1, 3}	: ”virtualmente” -> A={1, 3}, B={5}
iC 4	-> L={4, 5, 1, 3}	: ”virtualmente” -> A={1, 3}, B={5, 4}
acceder(2)	-> 5	: ”virtualmente” -> B[0]

P2:

Un caso interesante para analizar este algoritmo, es cuando el arreglo de enteros está en orden decreciente, por lo tanto el elemento A[i] que se inserta, siempre se comparará con todos los padres de R, hasta llegar a la raíz, pero como el elemento insertado queda siendo el nuevo R y además la raíz, entonces no se tendrá que revisar ni un padre. El costo para cada inserción de este caso es constante.

Por el contrario, si el arreglo estuviera en orden creciente, se irían insertando los nuevos elementos A[i], como hijos derechos de R. Los costos de hacer el árbol cartesiano serían iguales al caso del arreglo decreciente, pero ¿Qué pasa si, para este segundo caso, se inserta un elemento A[i] menor que A[1]? El costo de insertar ese elemento sería i, ya que deberá compararse con todos los padres de R hasta llegar a la raíz.

En efecto, vemos que el algoritmo tiene inserciones costosas cuando se ingresa un elemento que es menor que R y es proporcional a la profundidad de R menos la de M. Aún así, luego de hacer una inserción costosa, las siguientes no vuelven a ser tan costosas, ya que R vuelve a quedar más "arriba" en el árbol, y la diferencia de altura con M no es tanta.

Este caso se puede reducir el problema al stack multipop, siendo una inserción *barata* como un push y una inserción *costosa* como un multipop. Las inserciones *baratas* serán las que tienen costo constate, es decir cuando R=M (casos con arreglo creciente o decreciente). Las inserciones *caras* serán todas las que dependen de la altura de R y M, es decir, cuando se tiene que hacer más de una comparación para poder insertar A[i].

Si se deseara hacer un análisis por potencial, la función potencial sería fi=profundidad(R) – profundidad(M). "Cobraríamos la entrada y la salida" al principio, es decir, las inserciones baratas tendrían un Ci = 1 y deltaFi = 1, sumando 2 a la función potencial; por otro lado, las inserciones caras tendrían un Ci = -deltaFi = profundidad(R) – profundidad(M), resultando en un costo amortizado de O(2n). Por lo tanto, este algoritmo es O(n).

P3:

1) Conviene una cola de fibonacci, ya que principalmente se requiere unir colas y se podrían hacer en tiempo constante, muy superior a las otras técnicas. Ya que será una aplicación en tiempo real, sería algo absurdo querer extraer el mínimo, por lo que la "debilidad" de la cola de fibonacci no estaría afectando el desempeño de la aplicación. Para todas las demás operaciones, fibonacci es la mejor opción. *Hay que destacar, que la cola binomial podría brindar mejor orden de los datos que fibonacci, pero aún así se perdería la posibilidad de unir en costo constante.
2) La mejor opción sería una cola binomial. Será mejor que un heap, ya que se requieren unir colas de vez en cuando. Será mejor que una cola de fibonacci, ya que se podrían hacer varias extractMin caros, debido a inserciones constantes.
3) Ya que sólo se requiere un heapify y luego extraer mínimos repetidamente, podría servir tanto un heap como una cola binomial, ya que ambas técnicas tienen los mismos costos en esas operaciones. Por otro lado, para la cola de fibonacci, el primer extractMin sería muy caro, dejando al heap o a la cola binomial como las mejores opciones.