ESTRATEGIAS DE PROGRAMACIÓN
Y
ESTRUCTURAS DE DATOS

MEMORIA DE PRÁCTICA

Carlos Alberto Piñero Olanda

ÍNDICE

1. Preguntas teóricas

Sección 2.1

1. En un array disperso habrá un número de elementos equivalente al número real de elementos presentes, por lo tanto este debe ser el valor que devolverá el método size(). El método size() de Collection sólo devuelve el propio tamaño, pues no tiene implementado cuándo cambia su valor, por lo que depende de cada una de sus subclases cómo se hace. En cualquier caso, el tamaño del array disperso aumentará para la operación set() cuando se inserte un valor bajo un índice no presente y disminuirá para la operación remove() cuando en efecto exista un IndexedPair cuyo índice coincida con la posición buscada. Ya depende de cada implementación:

—Para SparseArraySequence, simplemente podemos usar un código análogo a los de las otras implementaciones de secuencias ya existentes.

—Para SparseArrayBTree, debemos cambiarlo porque el método size() de BTree depende del número de nodos del árbol binario, pero en nuestra implementación sólo debemos tener en cuenta el número de elementos existentes, esto es, no tener en cuenta los nodos vacíos.

2. Una colección, el objeto representado por la interfaz Collection, es un grupo de objetos sin otra relación entre ellos que la propia pertenencia al grupo. Sequence es una extensión de Collection que impone un orden al grupo de objetos. Los arrays dispersos, como arrays, necesitan tener un concepto implícito de orden para la operación de inserción en una posición determinada, por ejemplo, por lo que Sequence es la elección adecuada.

Sección 2.2

1. La lista, esto es, la clase List. No se pueden usar las otras dos, pilas y colas, Stack y Queue, porque en ambas sólo los elementos situados en primer lugar respectivamente pueden obtenerse, y además sólo pueden añadirse nuevos elementos en lo alto o al final, respectivamente para cada clase. Para realizar cualquiera de las operaciones anteriores en otra posición, sería obligatorio un método destructivo para llegar a este y reconstruir de nuevo la pila o cola. Algorítmicamente, es costoso e impráctico, y teniendo ya List está disponible es innecesariamente complicado.

2. Es mejor que los índices estén ordenados, en orden creciente preferiblemente, por dos razones:

—El iterador de índices pide que los índices estén en orden creciente, por lo que estando ya ordenados así sería más sencillo y menos costoso al encontrar en el primer bucle los índices ya ordenados.

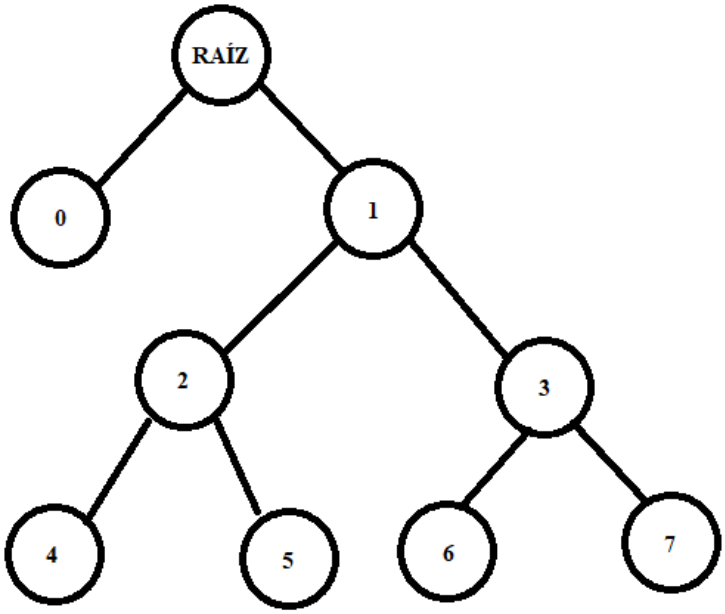
—Si no tuvieran ningún orden, por otro lado, causaría que las otras operaciones estuvieran en el peor caso para los costes algorítmicos cuando la posición buscada no corresponda a ningún Indexedpair presente: get, set y remove estarían obligados a recorrer el bucle entero en esa situación. Si están ordenados, por el contrario, basta con controlar que el siguiente índice del IndexedPair presente sea mayor al buscado.

Por otro lado, lo más probable es que la numeración de los elementos del array disperso corresponda a un orden real práctico, así que es conveniente respetarlo.

3. No debería, porque, como hemos explicado en una respuesta anterior, Collection no contempla el orden: es una clase ciega al que impongamos en nuestro array disperso, que en realidad se define en las subclases correspondientes.

Sección 2.3

1. La figura lo muestra. Aunque sólo indiquemos el índice, se supone que hay elementos en todos ellos.



Los elementos se ordenan de modo que el primero se sitúa en el hijo izquierdo del árbol y el siguiente en el derecho. Posteriormente, esto se repite para cada uno de estos dos hijos, consecuentemente, excepto para la posición 0. Para encontrar los objetos en orden creciente de índice, es preciso visitar los nodos situados a una altura determinada empezando por el situado más a la izquierda y avanzando uno a uno hacia la derecha. Esto en un árbol se llama búsqueda en anchura, por lo que este sería el modo de recorrerlo.

2. Casi todos los métodos funcionarían del mismo modo, pues no comprueban el índice del IndexedPair, excepto indexIterator, cuyo código debería controlar a qué índice corresponde y sería, pues, más complicado por tener ahora que calcularlo.

2. Diseño de las clases

El diseño de las clases es el que sigue. En la sección 3 se explicarán las partes desactivadas mediante comentario.

2.1. SparseArraySequence

```
package es.uned.lsi.eped.pract2021_2022;

import es.uned.lsi.eped.DataStructures.IteratorIF;
import es.uned.lsi.eped.DataStructures.List;
import es.uned.lsi.eped.DataStructures.Queue;
import es.uned.lsi.eped.DataStructures.QueueIF;
import es.uned.lsi.eped.DataStructures.Sequence;

public class SparseArraySequence<E>  extends Sequence<E>  implements SparseArrayIF<E> {

	protected List<IndexedPair<E>> list;

	public SparseArraySequence() {
		super();
		list = new List<IndexedPair<E>>();
	}

	public SparseArraySequence(SparseArraySequence<E> s) {
		super(s);
	}

	@Override
	public void delete(int pos) {
		int i = 1;
		IteratorIF<Integer> iter = this.indexIterator();
		while (iter.hasNext()) {
			int indice = iter.getNext();
			if (indice == pos) {
				list.remove(i);
				this.size--;
			} else if (indice > pos) {break;}
			i++;
		}
	}

	@Override
	public IteratorIF<Integer> indexIterator() {
		QueueIF<Integer> queue = new Queue<Integer>();
		IteratorIF<IndexedPair<E>> iterLista = list.iterator();
		while (iterLista.hasNext()) {
			queue.enqueue(iterLista.getNext().getIndex());
		}
		return queue.iterator();
	}

	@Override
	public void set(int pos, E elem) {
		IndexedPair<E> iP = new IndexedPair<E>(pos, elem);
//		long total = 0;
//		for (int u = 1; u <= 100000; u++) {
		int i = 1;
```

```
			IteratorIF<Integer> iter = this.indexIterator();
//			long inicio = System.nanoTime();
			while (iter.hasNext()) {
				int indice = iter.getNext();
				if (indice == pos) {
					list.set(i, iP);
					break;
				} else if (indice > pos) {
					list.insert(i, iP);
					this.size++;
					break;
				}
				i++;
			}
			if (i > this.size) {
				list.insert(i, iP);
				this.size++;
			}
//			long fin = System.nanoTime();
//			total = total + fin - inicio;
//			}
//			total = (total) / 100000;
//			System.out.println("Set para la posición " + pos + " y con " +  (this.size
- 1) + " elementos ha tardado " + total + "ns");

		}

		@Override
		public E get(int pos) {
			E resultado = null;
//			long total = 0;
//			for (int u = 1; u <= 100000; u++) {
			int i = 1;
			IteratorIF<Integer> iter = this.indexIterator();
//			long inicio = System.nanoTime();
			while (iter.hasNext()) {
				int indice = iter.getNext();
				if (indice == pos) {
					resultado = list.get(i).getValue();
					break;
				} else if (indice > pos) {break;}
				i++;
			}
//			long fin = System.nanoTime();
//			total = total + fin - inicio;
//			}
//			total = (total) / 100000;
//			System.out.println("Get para la posición " + pos + " y con " +
(this.size) + " elementos ha tardado " + total + "ns");
			return resultado;
		}

		@Override
		public IteratorIF<E> iterator() {
			QueueIF<E> queue = new Queue<E>();
			IteratorIF<IndexedPair<E>> iterLista = list.iterator();
			while (iterLista.hasNext()) {
				queue.enqueue(iterLista.getNext().getValue());
			}
```

```
		return queue.iterator();
	}

	public boolean contains(E e) {
		IteratorIF<E> iter = this.iterator();
		while (iter.hasNext()) {
			if (iter.getNext().equals(e)) {
				return true;
			}
		}
		return false;
	}

	public void clear() {
		this.list.clear();
		this.size = 0;
	}

}
```

2.2. SparseArrayBtree

```
package es.uned.lsi.eped.pract2021_2022;

import es.uned.lsi.eped.DataStructures.BTreeIF;
import es.uned.lsi.eped.DataStructures.BTree;
import es.uned.lsi.eped.DataStructures.StackIF;
import es.uned.lsi.eped.DataStructures.Stack;
import es.uned.lsi.eped.DataStructures.IteratorIF;
import es.uned.lsi.eped.DataStructures.Queue;
import es.uned.lsi.eped.DataStructures.QueueIF;

public class SparseArrayBTree<E> extends BTree<E> implements SparseArrayIF<E> {

	protected BTreeIF<IndexedPair<E>> btree;

	public SparseArrayBTree() {
		super();
		this.btree = new BTree<IndexedPair<E>>();
	}

	private StackIF<Boolean> num2bin(int n) {
		Stack<Boolean> salida = new Stack<Boolean>();
		if ( n == 0 ) {
			salida.push(false);
		} else {
			while ( n != 0 ) {
				salida.push((n % 2) == 1);
				n = n / 2;
			}
		}
		return salida;
	}

	@Override
	public IteratorIF<E> iterator() {
		QueueIF<E> queue = new Queue<E>();
		BTreeIF<IndexedPair<E>> btreeLocal = this.btree;
		if ( this.size > 0 ) {
			QueueIF<BTreeIF<IndexedPair<E>>> auxQ = new
Queue<BTreeIF<IndexedPair<E>>>();
			auxQ.enqueue(btreeLocal);
			while ( ! auxQ.isEmpty() ) {
				BTreeIF<IndexedPair<E>> cBT = auxQ.getFirst();
				if ( cBT.getLeftChild() != null ) {
					auxQ.enqueue(cBT.getLeftChild());
					if (cBT.getLeftChild().getRoot() != null) {

	queue.enqueue(cBT.getLeftChild().getRoot().getValue());
					}
				}
				if ( cBT.getRightChild() != null ) {
					auxQ.enqueue(cBT.getRightChild());
					if (cBT.getRightChild().getRoot() != null) {

	queue.enqueue(cBT.getRightChild().getRoot().getValue());
					}
				}
				auxQ.dequeue();
```

```
                }
            }
            return queue.iterator();
        }

        @Override
public void set(int pos, E elem) {
//      long inicio = System.nanoTime();
//      for (int u = 1; u <= 1000000; u++) {
        StackIF<Boolean> salida = this.num2bin(pos);
        BTreeIF<IndexedPair<E>> btreeLocal = this.btree;
        while (!salida.isEmpty()) {
            if (salida.getTop()) {
                if (btreeLocal.getRightChild() == null) {
                    BTreeIF<IndexedPair<E>> btreeNuevo = new
BTree<IndexedPair<E>>();
                    btreeLocal.setRightChild(btreeNuevo);
                }
                btreeLocal = btreeLocal.getRightChild();
            } else {
                if (btreeLocal.getLeftChild() == null) {
                    BTreeIF<IndexedPair<E>> btreeNuevo = new
BTree<IndexedPair<E>>();
                    btreeLocal.setLeftChild(btreeNuevo);
                }
                btreeLocal = btreeLocal.getLeftChild();
            }
            salida.pop();
        }
        if (btreeLocal.getRoot() == null) {this.size++;}
        IndexedPair<E> iP = new IndexedPair<E>(pos, elem);
        btreeLocal.setRoot(iP);
//          }
//          long fin = System.nanoTime();
//          long tiempo = (fin - inicio) / 1000000;
//          System.out.println("Set para la posición " + pos + " y con " +  (this.size
- 1) + " elementos ha tardado " + tiempo + "ns");
        }

        @Override
        public E get(int pos) {
            E resultado = null;
//          long inicio = System.nanoTime();
//          for (int u = 1; u <= 1000000; u++) {
            StackIF<Boolean> salida = this.num2bin(pos);
            BTreeIF<IndexedPair<E>> btreeLocal = this.btree;
            while (!salida.isEmpty() && btreeLocal != null) {
                if (salida.getTop()) {
                    btreeLocal = btreeLocal.getRightChild();
                } else {
                    btreeLocal = btreeLocal.getLeftChild();
                }
                salida.pop();
            }
            if (btreeLocal != null) {
                if (btreeLocal.getRoot() != null) {
                    resultado = btreeLocal.getRoot().getValue();
                }
            }
```

```
//            }
//            long fin = System.nanoTime();
//            long tiempo = (fin - inicio) / 1000000;
//            System.out.println("Get para la posición " + pos + " y con " +  (this.size
- 1) + " elementos ha tardado " + tiempo + "ns");
            return resultado;
      }

      @Override
      public void delete(int pos) {
            StackIF<Boolean> salida = this.num2bin(pos);
            BTreeIF<IndexedPair<E>> btreeLocal = this.btree;
            int arbolesVacios = 0;
            while (!salida.isEmpty() && btreeLocal != null) {
                  if (salida.getTop()) {
                        btreeLocal = btreeLocal.getRightChild();
                  } else {
                        btreeLocal = btreeLocal.getLeftChild();
                  }
                  if (btreeLocal == null) {
                        break;
                  }
                  salida.pop();
                  if (btreeLocal.getNumChildren() < 2 && btreeLocal.getRoot() == null
&& !salida.isEmpty()
                              || btreeLocal.getNumChildren() == 0 &&
salida.isEmpty()) {
                        arbolesVacios++;
                  } else {

                        arbolesVacios = 0;
                  }
            }

            if (btreeLocal != null) {
                  if (btreeLocal.getRoot() != null) {
                        this.size--;
                        btreeLocal.setRoot(null);
                        if (arbolesVacios > 0) {
                              arbolesVacios--;
                              while (arbolesVacios > 0) {
                                    arbolesVacios--;
                                    pos = pos / 2;
                              }
                              StackIF<Boolean> salidaEliminar = this.num2bin(pos);
                              btreeLocal = this.btree;
                              while (salidaEliminar.size() > 1) {
                                    if (salidaEliminar.getTop()) {
                                          btreeLocal = btreeLocal.getRightChild();
                                    } else {
                                          btreeLocal = btreeLocal.getLeftChild();
                                    }
                              salidaEliminar.pop();
                              }
                              if (salidaEliminar.getTop()) {
                                    btreeLocal.setRightChild(null);
                              } else {
                                    btreeLocal.setLeftChild(null);
                              }
```

```
                        }
                    }
                }
        }


        @Override
        public IteratorIF<Integer> indexIterator() {
                QueueIF<Integer> queue = new Queue<Integer>();
                BTreeIF<IndexedPair<E>> btreeLocal = this.btree;
                if ( !this.isEmpty() ) {
                        QueueIF<BTreeIF<IndexedPair<E>>> auxQ = new
Queue<BTreeIF<IndexedPair<E>>>();
                        auxQ.enqueue(btreeLocal);
                        while ( ! auxQ.isEmpty() ) {
                                BTreeIF<IndexedPair<E>> cBT = auxQ.getFirst();
                                if ( cBT.getLeftChild() != null ) {
                                        auxQ.enqueue(cBT.getLeftChild());
                                        if (cBT.getLeftChild().getRoot() != null) {

        queue.enqueue(cBT.getLeftChild().getRoot().getIndex());
                                                }
                                        }
                                if ( cBT.getRightChild() != null ) {
                                        auxQ.enqueue(cBT.getRightChild());
                                        if (cBT.getRightChild().getRoot() != null) {

        queue.enqueue(cBT.getRightChild().getRoot().getIndex());
                                                }
                                        }
                                auxQ.dequeue();
                        }
                }
                return queue.iterator();
        }

        public boolean isEmpty() {
                return this.size == 0;
        }

        public int size() {
                return this.size;
        }

        public boolean contains(E e) {
                IteratorIF<E> iter = this.iterator();
                while (iter.hasNext()) {
                        if (iter.getNext().equals(e)) {
                                return true;
                        }
                }
                return false;
        }

        public void clear() {
                this.btree.clear();
                this.size = 0;
        }
}
```

3. Estudio empírico de los costes

Sección 6

1. El tamaño del problema es distinto para cada implementación.

—En el caso de SparseArraySequence, el tamaño del problema es el número de elementos realmente presentes en el mismo. Es obvio, cuando se considera que cualquiera de los dos métodos, get y set, tendrá que viajar de nodo en nodo hasta que encuentre su posición. El peor caso posible es cuando el índice corresponda al último lugar, es decir, sea el más alto, por lo que calculemos el coste:

·Método get:

```
public E get(int pos) {
	E resultado = null;
//	long total = 0;
//	for (int u = 1; u <= 100000; u++) {
	int i = 1;
	IteratorIF<Integer> iter = this.indexIterator();
//	long inicio = System.nanoTime();
	while (iter.hasNext()) {
		int indice = iter.getNext();
		if (indice == pos) {
			resultado = list.get(i).getValue();
			break;
		} else if (indice > pos) {break;}
		i++;
	}
//	long fin = System.nanoTime();
//	total = total + fin - inicio;
//	}
//	total = (total) / 100000;
//	System.out.println("Get para la posición " + pos + " y con " +  (this.size) + "
elementos ha tardado " + total + "ns");
	return resultado;
}
```

Analizaremos sus partes: el principio, el bucle while y el final. Las partes comentadas se ignorarán en todos los análisis, así como los métodos de valor constante.

1. Es de coste constante: cualquier declaración, comparación u operación aritmética lo es, pues ninguna depende del tamaño del problema.

2. Constante.

3. El idexIterator es así:

```
public IteratorIF<Integer> indexIterator() {
	QueueIF<Integer> queue = new Queue<Integer>();
	IteratorIF<IndexedPair<E>> iterLista = list.iterator();
	while (iterLista.hasNext()) {
		queue.enqueue(iterLista.getNext().getIndex());
	}
	return queue.iterator();
}
```

Las líneas antes y después del while son constantes. El coste del propio bloque se calcula como el producto del coste del número de vueltas por el máximo entre el coste de la comprobación de si sigue el bucle y el cuerpo del mismo:

$O(v(n))\cdot max\{O(c_e), O(c_S)\}$

El primer operando es lineal, pues se dan tantas vueltas como elementos hay en el peor de los casos:

$O(v(n)) = O(n)$

El segundo y el tercero son constante, pues son simples declaraciones:

$O(c_e) = O(c_S)= O(1)$

Luego:

$O(n)\cdot\{maxO(1), O(1)\} = O(n)\cdot O(1) = O(n)$

Es lineal.

4. El coste de un bucle while se calcula así, como el producto del coste del número de vueltas por el máximo entre el coste de la comprobación de si sigue el bucle y el cuerpo del mismo:

$O(v(n))\cdot max\{O(c_e), O(c_S)\}$

Ya hemos explicado que el primer operando es lineal. El segundo es constante, pues el comando sólo comprueba el valor de una variable booleana:

$O(c_e) = O(1)$

El tercero tiene tres partes: una llamada a un método una línea de incremento de variable de coste constante y un condicional if. Este se calcula como el máximo entre sus tres partes, comprobación, salida 1 y salida 2:

$max\{O(c_e), O(c_{S1}), O(c_{S2})\}$

Como aquí tenemos un if else, se incluye una segunda comprobación. De momento, el cuerpo del bucle tiene un coste:

$O(c_S) = O(1) + max\{O(c_{e1}), O(c_{S1}), O(c_{e2}), O(c_{S2})\} = O(1) + max\{O(1), O(c_{S1}), O(1), O(1)\}$

En la salida 1 se llama a otro método, que es el siguiente:

```
protected NodeSequence getNode(int i){
	NodeSequence node = this.firstNode;
	for ( int aux = 1 ; aux < i ; aux++ ) {
		node = node.getNext();
	}
	return node;
}
```

Dos líneas de coste constante y un bucle for, cuyo coste es el máximo entre el coste de iniciar el contador y el producto del número de vueltas por el máximo, a su vez, entre el coste de la condición que determina sigue el bucle, el contenido del bucle y la condición de incrementar el contador. Así que el bucle tiene un coste:

$\max\{O(c_{ini}), O(v(n))\cdot\max\{O(c_e), O(c_S), O(c_{inc})\}\} = \max\{O(1), O(n)\cdot\max\{O(1), O(1), O(1)\}\} = \max\{O(1), O(n)\cdot O(1)\} = \max\{O(1), O(n)\} = O(n)$

Lineal, por tanto. Volviendo a la expresión anterior:

$O(c_S) = O(1) + \max\{O(1), O(n), O(1), O(1)\} = O(c_S) = O(1) + O(n) = O(n)$

Es lineal también. Antes de continuar el análisis, es necesario aclarar: los dos condicionales if contienen el comando break. Este provoca que el bucle termine, lo que es muy significativo, pues en este caso el coste de los condicionales no se multiplicará a las vueltas que dé el bucle, ya que sólo en la última vuelta se llamará. Se puede visualizar el bucle como un coste lineal de n – 1 vueltas más, en su caso, el coste del condicional de coste asimismo lineal. En este caso, hemos de reescribir el coste algorítmico del bucle como la suma del producto del coste de las vueltas por la declaración que sí se presenta en cada bucle y el máximo entre ambos condicionales:

$O(n)\cdot\max\{O(1)\} + \max\{O(n), O(1)\} = O(n) + O(n) = O(n)$

5. El coste es constante.

Total: $O(1) + O(1) + O(n) + O(n) + O(1) = O(n)$

Es decir, su coste es lineal.

·Método set:

```
public void set(int pos, E elem) {
	IndexedPair<E> iP = new IndexedPair<E>(pos, elem);
//	long total = 0;
//	for (int u = 1; u <= 100000; u++) {
	int i = 1;
	IteratorIF<Integer> iter = this.indexIterator();
//	long inicio = System.nanoTime();
	while (iter.hasNext()) {
		int indice = iter.getNext();
		if (indice == pos) {
			list.set(i, iP);
			break;
		} else if (indice > pos) {
			list.insert(i, iP);
			this.size++;
			break;
		}
		i++;
	}
	if (i > this.size) {
		list.insert(i, iP);
		this.size++;
	}
//	long fin = System.nanoTime();
```

```
//     total = total + fin - inicio;
//     }
//     total = (total) / 100000;
//     System.out.println("Set para la posición " + pos + " y con " +  (this.size - 1) +
" elementos ha tardado " + total + "ns");

}
```

Como antes, lo separamos en tres partes:

1. Las dos declaraciones iniciales son constantes.

2. El indexIterator es lineal.

3. Este bucle es parecido al de get, así que lo único que señalaremos es cómo son los métodos nuevos llamados en los condicionales:

```
public void set(int pos, E e) {
	NodeSequence node = getNode(pos);
	node.setValue(e);
}
```

Ya hemos visto que getNode es lineal. Respecto a insert:

```
public void insert(int pos, E elem) {
	NodeSequence newNode = new NodeSequence(elem);
	if(pos==1){
		newNode.setNext(this.firstNode);
		this.firstNode = newNode;
	}else{
		NodeSequence previousNode = getNode(pos-1);
		NodeSequence nextNode = previousNode.getNext();
		previousNode.setNext(newNode);
		newNode.setNext(nextNode);
	}
	this.size++;
}
```

Las declaraciones fueran del condicional son constantes. Respecto a este, como ya hemos analizado antes los métodos:

$\max\{O(c_e), O(c_{S1}), O(c_{S2})\} = \max\{O(1), O(1), O(n) + O(1) + O(1) + O(1)\} =$
$\max\{O(1), O(1), O(n)\} = O(n)$

También es lineal. El condicional vale entonces:

$\max\{O(c_{e1}), O(c_{S1}), O(c_{e2}), O(c_{S2})\} = \max\{O(1), O(n), O(1), O(n)\} = O(n)$

En este caso, como volvemos a usar el comando break, el coste del bucle vale:

$O(n)\cdot\max\{O(1)\} + \max\{O(n), O(n)\} = O(n) + O(n) = O(n)$

4. Este condicional usa métodos ya vistos, por lo que:

$\max\{O(c_{e1}), O(c_{S1})\} = \max\{O(1), O(n)\} = O(n)$

Total: O(1) + O(n) + O(n) + O(n) = O(n)

También lineal.

—En el caso de SparseArrayBTree, como su estructura no obliga a recorrer todos los elementos, el tamaño del problema es distinto: es el índice del IndexedPair. Cuanto más alto sea el índice, más nodos habrá enraizados y la búsqueda llevará mayor tiempo. A diferencia del caso anterior, que haya o no más elementos no afecta en absoluto, incluso en los nodos de la rama, pues existirían de todos modos mientras haya un elemento en la hoja. El peor caso posible es, se deduce, cuando se busca el término cuyo índice precise la expresión en bits más larga. De aquí se deduce que los costes deben ser logarítmicos, pues el número de nodos siempre coincide con la parte entera del logaritmo binario del índice. De todos modos, el cálculo es como sigue:

·Método get:

```
public E get(int pos) {
	E resultado = null;
//	long inicio = System.nanoTime();
//	for (int u = 1; u <= 1000000; u++) {
	StackIF<Boolean> salida = this.num2bin(pos);
	BTreeIF<IndexedPair<E>> btreeLocal = this.btree;
	while (!salida.isEmpty() && btreeLocal != null) {
		if (salida.getTop()) {
			btreeLocal = btreeLocal.getRightChild();
		} else {
			btreeLocal = btreeLocal.getLeftChild();
		}
		salida.pop();
	}
	if (btreeLocal != null) {
		if (btreeLocal.getRoot() != null) {
			resultado = btreeLocal.getRoot().getValue();
		}
	}
//	}
//	long fin = System.nanoTime();
//	long tiempo = (fin - inicio) / 1000000;
//	System.out.println("Get para la posición " + pos + " y con " +  (this.size - 1) +
" elementos ha tardado " + tiempo + "ns");
	return resultado;
}
```

Como antes, analizaremos parte por parte:

1. El coste es constante: O(1)

2. El método num2bin es como sigue:

```
private StackIF<Boolean> num2bin(int n) {
	Stack<Boolean> salida = new Stack<Boolean>();
	if ( n == 0 ) {
		salida.push(false);
	} else {
		while ( n != 0 ) {
```

```
                        salida.push((n % 2) == 1);
                        n = n / 2;
                  }
            }
            return salida;
}
```

$O(1) + \max\{O(c_e), O(c_{S1}), O(v(n))\cdot\max\{O(c_e), O(c_S)\}\} + O(1) =$
$= \max\{O(1), O(1), O(n)\cdot\max\{O(1), O(1)\}\}\} = O(n)\cdot O(1) = O(n) = O(\log_2 pos)$

Está claro, pues, que su coste es logarítmico recordando cuál es el tamaño del problema aquí.
3. Constante.

4. Otro bucle while:

$O(v(n))\cdot\max\{O(c_e), O(c_S)\} = O(n)\cdot\max\{O(1), \max\{O(c_e), O(c_{S1}), O(c_{S2})\} + O(1)\} =$
$O(n)\cdot\max\{O(1), \max\{O(1), O(1), O(1)\} + O(1)\} = O(n)\cdot\max\{O(1), O(1)\} = O(n)\cdot O(1) =$
$O(n) = O(\log_2 pos)$

De nuevo logarítmico.

5. Dos condiciones anidadas sin alternativas, su coste es:

$\max\{O(c_e), O(c_{S1})\} = \max\{O(1), \max\{O(c_e), O(c_{S1})\}\} = \max\{O(1), \max\{O(1), O(1)\}\} = O(1)$

6. Constante.

Total: $O(1) + O(\log_2 pos) + O(\log_2 pos) + O(1) + O(1) = O(\log_2 pos)$

Por lo tanto, el coste es logarítmico.

·Método set:

```
public void set(int pos, E elem) {
//     long inicio = System.nanoTime();
//     for (int u = 1; u <= 1000000; u++) {
       StackIF<Boolean> salida = this.num2bin(pos);
       BTreeIF<IndexedPair<E>> btreeLocal = this.btree;
       while (!salida.isEmpty()) {
             if (salida.getTop()) {
                   if (btreeLocal.getRightChild() == null) {
                         BTreeIF<IndexedPair<E>> btreeNuevo = new
BTree<IndexedPair<E>>();
                         btreeLocal.setRightChild(btreeNuevo);
                   }
                   btreeLocal = btreeLocal.getRightChild();
             } else {
                   if (btreeLocal.getLeftChild() == null) {
                         BTreeIF<IndexedPair<E>> btreeNuevo = new
BTree<IndexedPair<E>>();
                         btreeLocal.setLeftChild(btreeNuevo);
                   }
                   btreeLocal = btreeLocal.getLeftChild();
             }
             salida.pop();
       }
```

```
        if (btreeLocal.getRoot() == null) {this.size++;}
        IndexedPair<E> iP = new IndexedPair<E>(pos, elem);
        btreeLocal.setRoot(iP);
//            }
//            long fin = System.nanoTime();
//            long tiempo = (fin - inicio) / 1000000;
//            System.out.println("Set para la posición " + pos + " y con " +  (this.size
- 1) + " elementos ha tardado " + tiempo + "ns");
}
```

Es muy similar al anterior:

1. Ya sabemos de get() que es $O(\log_2 pos)$.

2. $O(1)$

3. Este bucle while es más complejo que el análogo de get:

$O(v(n))\cdot\max\{O(c_e), O(c_S)\} = O(n)\cdot\max\{O(1), \max\{O(c_e), O(c_{S1}), O(c_{S2})\} + O(1)\} =$
$O(n)\cdot\max\{O(1), \max\{O(1), \max\{O(c_e), O(c_S)\} + O(1), \max\{O(c_e), O(c_S)\} + O(1)\} + O(1)\} =$
$O(n)\cdot\max\{O(1), \max\{O(1), O(1)\} + O(1), \max\{O(1), O(1)\} + O(1)\} + O(1)\} =$
$O(n)\cdot\max\{O(1), O(1) + O(1)\} = O(n)\cdot\max\{O(1), O(1)\} = O(n)\cdot O(1) = O(n) = O(\log_2 pos)$

Logarítmico de nuevo.

4. El condicional es:

$\max\{O(c_e), O(c_{S1})\} = \max\{O(1), \max\{O(c_e), O(c_{S1})\}\} = \max\{O(1), \max\{O(1), O(1)\}\} = O(1)$

5. Constante

6. Constante

Total: $O(\log_2 pos) + O(1) + O(\log_2 pos) + O(1) + O(1) = O(\log_2 pos)$

Por lo tanto, el coste es también logarítmico.

2. En primer lugar, explicaremos cómo hemos diseñado el estudio empírico. Básicamente, hemos creado dos ficheros del bloc de notas de Windows, en el que hemos escrito una serie de órdenes get y set, siguiendo la explicación del enunciado para la clase main del fichero de prácticas. Estas órdenes se hallan en la carpeta titulada «Juegos de prueba» y tienen nombres específicos para cada implementación.

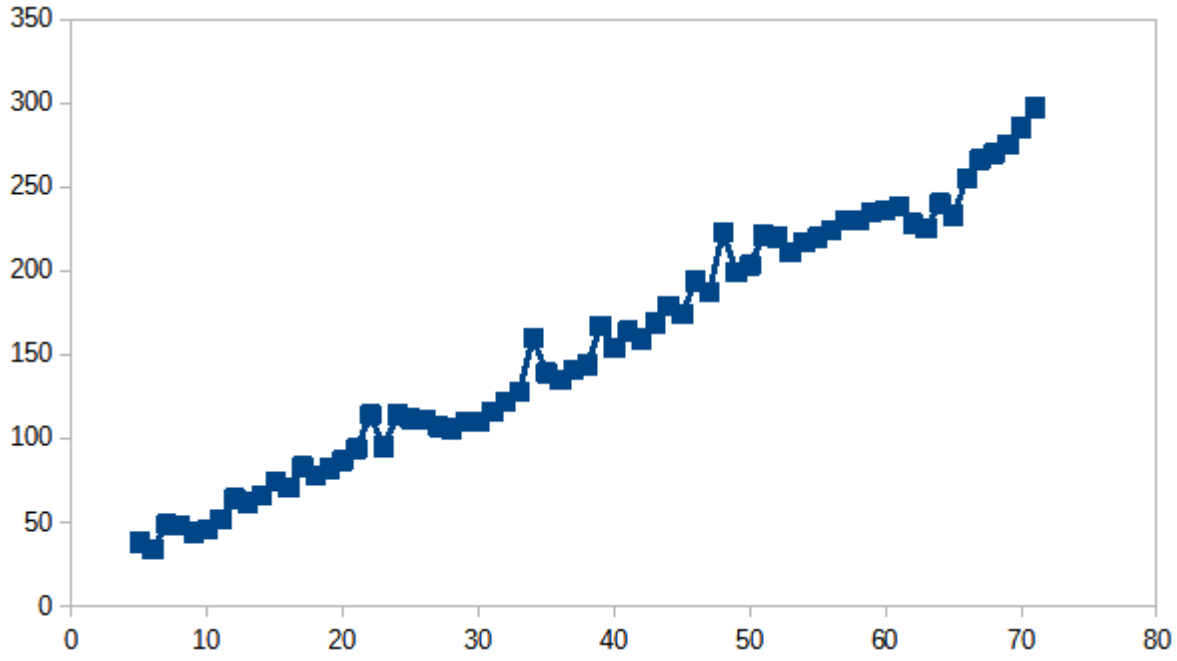
Luego, en el código hemos incluido líneas para medir el tiempo cuando se ejecutaban las órdenes. Estas son las líneas comentadas, como sólo podemos entregar los ficheros de las dos clases, hemos decidido dejarlos comentados. Estas medidas se hacían con bucles for porque, dependiendo de la actividad del PC, el tiempo de ejecución puede variar. Además, se han descartado las primeras ejecuciones porque, por sistema, dan tiempos notoriamente mayores que los demás.

Los resultados obtenidos se expondrán junto a los gráficos resultantes para cada implementación.

—Para SparseArraySequence, los datos y gráficos han sido:

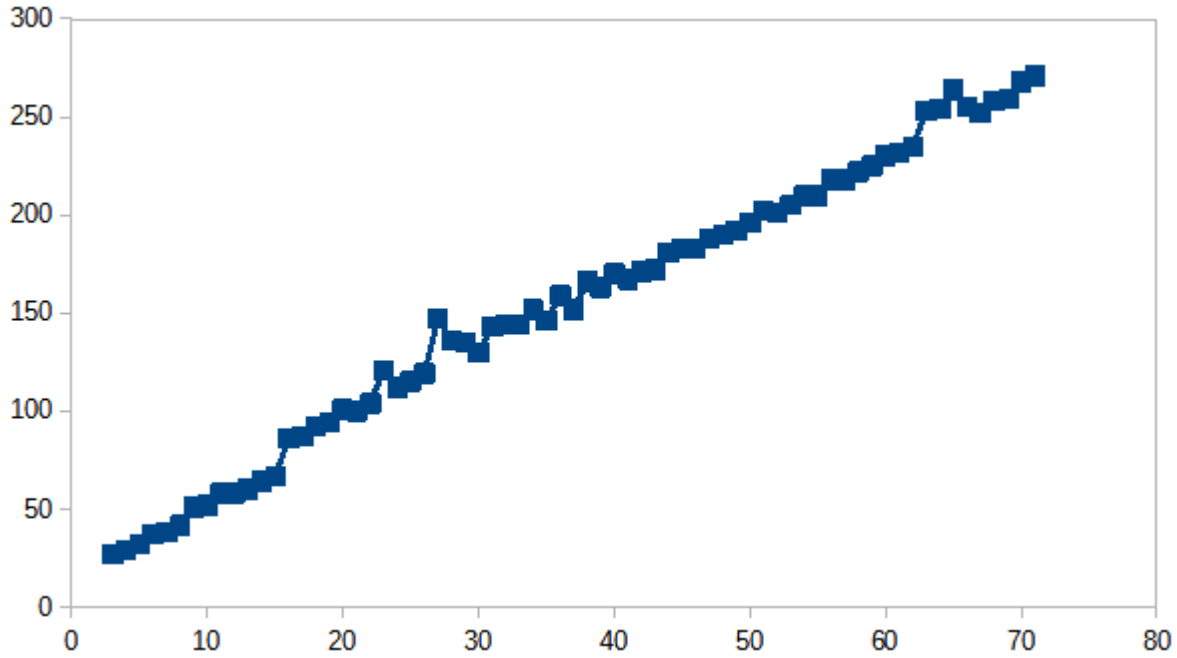
·Set

elementos	tiempo	elementos	tiempo	elementos	tiempo
0	133	24	114	48	223
1	261	25	112	49	199
2	216	26	111	50	203
3	47	27	107	51	221
4	50	28	106	52	220
5	38	29	110	53	211
6	34	30	110	54	217
7	49	31	116	55	220
8	48	32	122	56	224
9	44	33	128	57	230
10	46	34	160	58	230
11	52	35	139	59	235
12	64	36	135	60	236
13	62	37	141	61	238
14	66	38	144	62	228
15	74	39	167	63	225
16	71	40	154	64	240
17	83	41	164	65	233
18	78	42	159	66	255
19	82	43	169	67	266
20	87	44	179	68	270
21	94	45	174	69	275
22	114	46	194	70	285
23	95	47	187	71	297



·Get

elementos	tiempo	elementos	tiempo	elementos	tiempo
1	78	25	115	49	192
2	132	26	119	50	196
3	27	27	147	51	202
4	29	28	136	52	201
5	32	29	135	53	205
6	37	30	130	54	210
7	38	31	143	55	210
8	42	32	144	56	218
9	51	33	144	57	218
10	52	34	152	58	222
11	58	35	146	59	225
12	58	36	159	60	230
13	60	37	152	61	232
14	64	38	166	62	235
15	67	39	163	63	253
16	86	40	170	64	254
17	87	41	167	65	264
18	92	42	171	66	255
19	94	43	172	67	252
20	101	44	181	68	258
21	100	45	183	69	259
22	104	46	183	70	268
23	121	47	188	71	271
24	112	48	190		

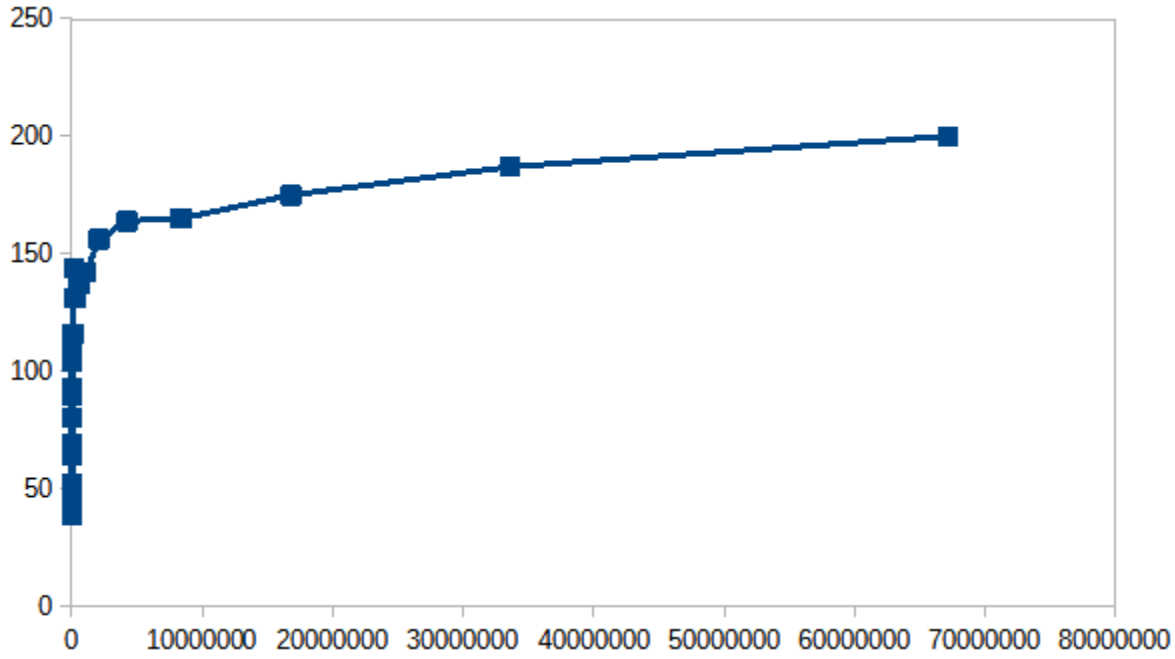


Ambos métodos tienen tendencia lineal, acorde a lo predicho teóricamente. Es obvio que se nota más cuando el número de elementos empieza a ser considerable, lo que insinúa que al principio acceder a algunos métodos supone mayor coste que el tamaño del problema.

—Para SparseArrayBtree:

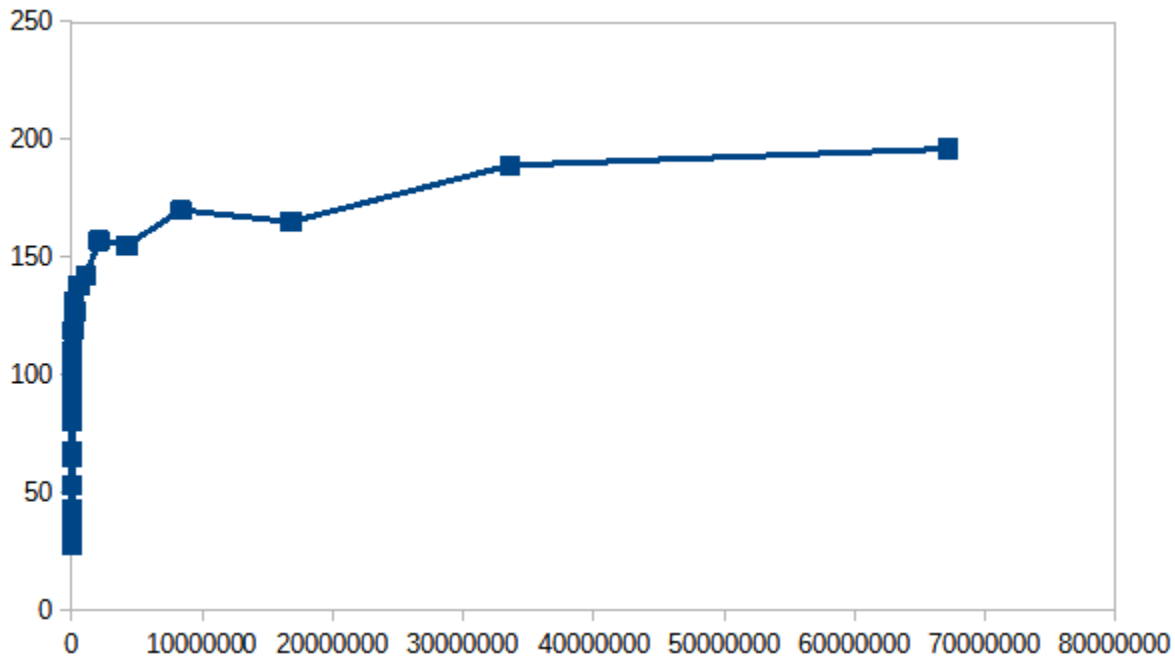
·Set:

posición	tiempo	posición	tiempo
0	49	16384	110
2	52	32768	113
4	45	65536	116
8	52	131072	144
16	39	262144	131
32	48	524288	137
64	69	1048576	142
128	64	2097152	156
256	66	4194304	164
512	104	8388608	165
1024	80	16777216	175
2048	90	33554432	187
4096	110	67108864	200
8192	93		



·Get:

posición	tiempo	posición	tiempo
0	31	16384	101
2	28	32768	110
4	35	65536	119
8	31	131072	131
16	37	262144	127
32	43	524288	138
64	53	1048576	142
128	65	2097152	157
256	67	4194304	155
512	80	8388608	170
1024	82	16777216	165
2048	90	33554432	189
4096	88	67108864	196
8192	94		



Ambas gráficas siguen claramente una forma logarítmica, caracterizada por un crecimiento inicial muy notorio para luego decaer progresivamente, aunque nunca dejan de ser crecientes. Como en el eje de abcisas aparece la posición, es obvio que nuestra suposición ha sido la correcta.

Por lo tanto, podemos concluir que los costes coinciden, aunque es importante darse cuenta de que en la ejecución hay otros factores que no hemos tenido en cuenta.