

Diseño de pruebas y análisis de complejidad

SANTIAGO HURTADO SOLIS A003625, CRISTIAN MORALES A00328064, JUAN
SEBASTIAN MORALES A00365920

Diseño de los casos de prueba Bank.

Clase	Metodo	Escenario	Valores Entrada	Resultado
StackTest	testStack	Se crea una Pila	Null	Se revisa que la pila creada no sea null
StackTest	testPush	Se crea una pila se agrega un elemento	"hola"	Se comprueba que la pila no esté vacía
StackTest	testPeek	Se crea una pila se agrega un elemento se utiliza el método peek	"hola"	Se comprueba que la pila no esté vacía además de que el elemento recuperado se igual al elemento que se agrego
StackTest	testPop	Se crea una pila se agrega un elemento se utiliza el método peek	"hola"	Se comprueba que la pila esté vacía además de que el elemento recuperado se igual al elemento que se agrego
StackTest	testIsEmpty	Se crea una pila y se agrega un elemento	"hola"	Se comprueba que la pila no esté vacía

Clase	Metodo	Escenario	Valores Entrada	Resultado
QueueTest	testEnqueue	Se crea una cola y se agrega un valor se utiliza el método enqueue	"hola"	Se revisa que la cola creada no sea null
QueueTest	testGetFirst	Se crea una cola se agrega un valor se utiliza el método getFirst	"hola"	Se revisa que el elemento obtenido que es el primero sea el que se agrego y que la cola este vacía
QueueTest	testConsult	Se creo una cola se agrega un valor se utiliza el método Consult	"hola"	Se revisa que el elemento consultado de la cola sea el agregado y que la cola no esté vacía

QueueTest	testDequeue	Se creo una cola se agrega un valor y se utiliza el método dequeue	"hola"	Se revisa que la cola no esté vacía y que el elemento recuperado sea el agregado
QueueTest	testIsEmpty	Se crea una cola se agrega un valor se utiliza el método isEmpty	"hola"	Se verifica que la cola no está vacía

Clase	Metodo	Escenario	Valores Entrada	Resultado
HashTableTest	testAdd	Se crea un hash table se agrega una tupla	"test, holaMundo"	Se verifica que el tamaño de la tabla sea diferente de 0
HashTableTest	testSize	Se crea una hash table y se agregan dos tuplas	"test,hola" "test2,mundo"	Se verifica que el tamaño de la tabla sea dos
HashTableTest	testGetList	Se crea una hash table y se agregan dos tuplas	"test,hola" "test2,mundo"	Se verifica que al obtener los valores con el metodo get list estos sean iguales a los valores ingresados en las tuplas
HashTableTest	testIsEmpty	Se crea una hash table	null	Se verifica que la hash table este vacia
HashTableTest	testRemove	Se crea una hash table Se agregan dos tuplas y se elimina una	"test,hola" "test2,mundo"	Se verifica que el tamaño de la tabla sea igual a 1.
HashTableTest	testResearch	Se crea una hash table Se agregan dos tuplas y se recupera cada valor	"test,hola" "test2,mundo"	Se compara los dos elementos y se comparan con los valores agregados

Clase	Metodo	Escenario	Valores Entrada	Resultado
PriorityQueueTest	testEnqueue	Se crea una cola, vacia y se agrega un elemento en ella.	"hola"	Se verifica que la cola de prioridad no este vacia
PriorityQueueTest	testGetFirst	Se crea una cola de prioridad, y se agrega un elemento en la primera posicion.	"hola"	se verifica que este nos devuelve ese mismo elemento en la primera posicion.
PriorityQueueTest	testConsult	Se crea una cola de prioridad , e agrega un elemento.	"hola"	Se comprueba que el elemento recuperado se haya agregado, y se verifica ademas que la priority queue no este vacia.
PriorityQueueTest	testDequeue	Se crea una cola de prioridad, se agrega un elemento , este elemento se desencola.	"hola"	Se verifica que el elemento agregado se haya eliminado.
PriorityQueueTest	testIsEmpty	Se crea una cola de prioridad.	""hola"	Se verifica que la cola este vacia.

Clase	Metodo	Escenario	Valores Entrada	Resultado
ListSortsTest	testMergeSort	Escenario 1	"Cristian", "Morales", "1101", Tarjet.AHORROS, "123456", 20000, LocalDate.now() Santiago", "Hurtado", "4010", Tarjet.AHORROS, "123456", 10000, LocalDate.now());	Se verifica que se ordenen los clientes por cantidad de dinero en la cuenta

			"Juan", "Morales", "3210", Tarjet.AHORROS, "123456", 30000, LocalDate.now());	
ListSortsTest	testSelectionSort	Escenario 1	"Cristian", "Morales", "1101", Tarjet.AHORROS, "123456", 20000, LocalDate.now() Santiago", "Hurtado", "4010", Tarjet.AHORROS, "123456", 10000, LocalDate.now()); "Juan", "Morales", "3210", Tarjet.AHORROS, "123456", 30000, LocalDate.now());	Se verifica que se ordenen los clientes por nombre
ListSortsTest	testHeapSort	Escenario 1	"Cristian", "Morales", "1101", Tarjet.AHORROS, "123456", 20000, LocalDate.now() Santiago", "Hurtado", "4010", Tarjet.AHORROS, "123456", 10000, LocalDate.now()); "Juan", "Morales", "3210", Tarjet.AHORROS, "123456", 30000, LocalDate.now());	Se verifica que se ordenen los clientes por ID

ListSortsTest	testQuickSort	Escenario 1	"Cristian", "Morales", "1101", Tarjet.AHORROS, "123456", 20000, LocalDate.now() Santiago", "Hurtado", "4010", Tarjet.AHORROS, "123456", 10000, LocalDate.now()); "Juan", "Morales", "3210", Tarjet.AHORROS, "123456", 30000, LocalDate.now());	Se verifica que se ordenen los clientes por fecha de ingreso al banco
---------------	---------------	-------------	--	---

Escenario 1

```
cliente1 = new Client("Cristian", "Morales", "1101", Tarjet.AHORROS, "123456", 20000,
    LocalDate.now());
```

```
cliente2 = new Client("Santiago", "Hurtado", "4010", Tarjet.AHORROS, "123456", 10000,
    LocalDate.now());
```

```
cliente3 = new Client("Juan", "Morales", "3210", Tarjet.AHORROS, "123456", 30000, LocalDate.now());
```

Explicacion Resolucion Casos de prueba paso a paso por estructura de datos.

Stack

Metodo: push

Escenario y resolucion : Se crea una Pila se agrega el elemento String hola con el metodo Push se verifica con el metodo empty que esta este vacia se utiliza el assert equals esperando false porque la pila no esta vacia.

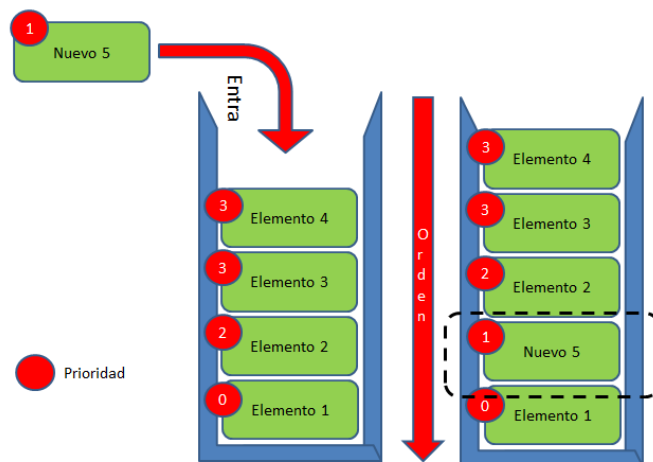
Metodo: pop

Escenario y resolucion: Se crea una Pila se agrega el elemento String hola con el metodo Push, se utiliza el metodo pop para sacar el elemento se verifica que el elemento sea igual al agregado y que la pila este vacia.

Queue

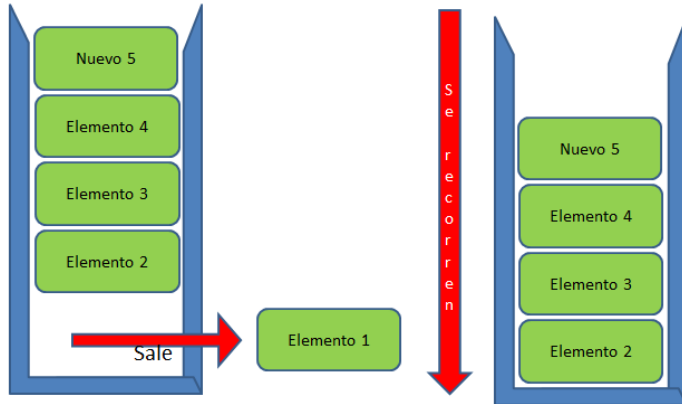
Metodo: queue

Escenario y resolucion: Se crea una cola se agrega un elemento String hola con el metodo enqueue y se verifica con el metodo isEmpty que la cola este vacia utilizando el metodo assert equals que se espera que sea false porque la cola posee un elemento.



Metodo: enqueue

Escenario y resolucion: Se crea una cola se agrega un elemento String hola con el metodo enqueue y se desencola con el metodo dequeue se verifica que el elemento sea igual al ingresado utilizando el metodo equals to el metodo assert equals que se espera que sea true porque son iguales.



Hashtable

Metodo: isEmpty

Escenario y resolucion: se crea una Hash table se utiliza el metodo isEmpty y se evalua el resultado con el metodo assert equals esperado true porque la tabla esta vacia

Metodo: size

Escenario y resolucion: se crea una hash table se agregan dos tuplas con el metodo add, se evalua el tamaño de la hashtable y se utiliza el metodo assert equals con numero esperado a 2 porque las tuplas que se agregaron fueron dos.

Priority queue

Metodo: enqueue

Escenario y resolucion : Se crea una cola de prioridad , y se encola un elemento String en ella.

Luego se verifica que la cola no este vacia con el metodo isEmpty() ,puesto que este elemento se agrego , posteriormente con el assertsEquals, que se espera que sea falso, porque ya habra un elemento en la priority queue.

Selection

```
private static <T> void exchange(List<T> array, int i, int j) {           // 3
    T aux = array.get(i);         // 1
    array.set(i, array.get(j));    // 1
    array.set(j, aux);             // 1
}

public static <T> void selectionSort(List<T> array, Comparator<T> comparator){ // n = array.length

    for (int i = 0; i < array.size(); i++) {                               // n+1
        int lower = i;                                                    // n
        for (int j = i+1; j < array.size(); j++) {                        // n(n+1)/2
            if(comparator.compare(array.get(j),array.get(lower))<0) {      // n(n+3)/2
                lower = j;                                                 // n(n+3)/2
            }
        }
        if(lower!=i) {                                                    // n
            exchange(array,lower,i);                                       // 3n
        }
    }
}

// SUPA
// (3n^2)/2 + 19n/2
// n(3n+19)/2 <= C*n^2
// O(n^2)
//
```

Complejidad Espacial

Selection Sort

	Complejidad espacial	
	variable	cantidad de valores atomicos
Entrada	array	n
salida	resultado	n
auxiliares	lower	1
	i	1
	j	1
resultado		2n+3

Complejidad Temporal Selection

Algoritmo	Linea de codigo	# veces que se repite
Exchange	T aux = array.get(i);	3
	array.set(i, array.get(j));	
	array.set(j, aux);	
	for (int i = 0; i < array.size(); i++)	n+1
	int lower = i;	n
Selection	for (int j = i+1; j < array.size(); j++)	$n(n+1)/2$
	if(comparator.compare(array.get(j),array.get(lower))<0)	$n(n+3)/2$
	lower = j;	$n(n+3)/2$
	if(lower!=i)	n
	exchange(array,lower,i);	3n
	TOTAL	$O(n^2)$

Heapsort

```

27
28 public static <T> void heapSort(List<T> array, Comparator<T> comparator){
29
30     int n = array.size()-1;
31     while(n>0) {
32         for (int i = (n-1)/2; i >=0; i--) {
33             heapifying(array,i,n,comparator);
34         }
35         exchange(array,0,n);
36         n--;
37     }
38 }
39
40 private static <T> void heapifying(List<T> array, int i,int size, Comparator<T> comparator) {
41     int left = 2*i +1;
42     int right = 2*i +2;
43     int max=i;
44     if(left<=size) {
45         if(comparator.compare(array.get(left), array.get(i))>0) {
46             max = left;
47         }
48     }
49     if(right<=size) {
50         if(comparator.compare(array.get(right), array.get(max))>0) {
51             max = right;
52         }
53     }
54     if(max!=i) {
55         exchange(array,i,max);
56         heapifying(array,max,size,comparator);
57     }
58 }

```

Complejidad temporal

Heap sort	# codigo	# veces que se repite
	int n = array.size()-1;	n
	while(n>0)	n+1
	for (int i = (n-1)/2; i >=0; i--)	n(n+1)
	heapifying(array,i,n,comparator);	$\lg n$
	exchange(array,0,n);	3
	n--;	n
Heapifying	int left = 2*i +1;	n
	int right = 2*i +2;	n
	int max=i;	n
	if(left<=size)	n
	if(comparator.compare(array.get(left), array.get(i))>0)	$n(n+1)/2$
	max = left;	n
	if(right<=size)	n
	if(comparator.compare(array.get(right), array.get(max))>0)	$n(n+1)/2$
	max = right;	n
	if(max!=i)	Log n
	exchange(array,i,max);	3
	heapifying(array,max,size,comparator);	Log n
Total		O (n log n)

Complejidad Espacial Heap Sort incluyendo variables del heapifying

	Complejidad espacial	
	variable	cantidad de valores atomicos
Entrada	array	n
salida	resultado	n
auxiliares	right	1
	size	1
	max	1
	left	1
	i	1
	n	1
resultado	2n+6	

Quicksort

```

179
180 public static <T> void quickSort(List<T> array, Comparator<T> comparator) {
181     quickSort(array, 0, array.size() - 1, comparator);
182 }
183
184 private static <T> void quickSort(List<T> array, int start, int end, Comparator<T> comparator) {
185     int firstStart = start;
186     int firstEnd = end;
187     if (end - start <= 0) {
188         return;
189     }
190     if (comparator.compare(array.get(start), array.get(end)) < 0) {
191         exchange(array, start, end);
192     }
193     }
194     } else {
195         int pivot = (start + end) / 2;
196         boolean order = false;
197         boolean alreadyOrdered = true;
198         while (order) {
199             boolean lower = false;
200             boolean upper = false;
201             if (comparator.compare(array.get(start), array.get(pivot)) < 0) {
202                 start++;
203             } else if (start == pivot) {
204                 lower = true;
205             }
206             if (comparator.compare(array.get(end), array.get(pivot)) < 0) {
207                 end--;
208             } else if (end == pivot) {
209                 upper = true;
210             }
211             if (lower && upper) {
212                 if (pivot == end) {
213                     pivot = start;
214                 } else if (pivot == start) {
215                     pivot = end;
216                 }
217             }
218             alreadyOrdered = false;
219         }
220         exchange(array, end, start);
221         if (start == pivot) {
222             start++;
223         }
224         if (end == pivot) {
225             end--;
226         }
227     }
228     order = end <= start && end == pivot;
229     }
230     if (!alreadyOrdered) {
231         quickSort(array, firstStart, pivot - 1, comparator);
232         quickSort(array, pivot + 1, firstEnd, comparator);
233     }
234 }
235 }

```

$$t(n) = \begin{cases} 0 & \text{Si } n = 1 \\ 2t(n/2) + n & \text{Si } n > 1 \end{cases}$$

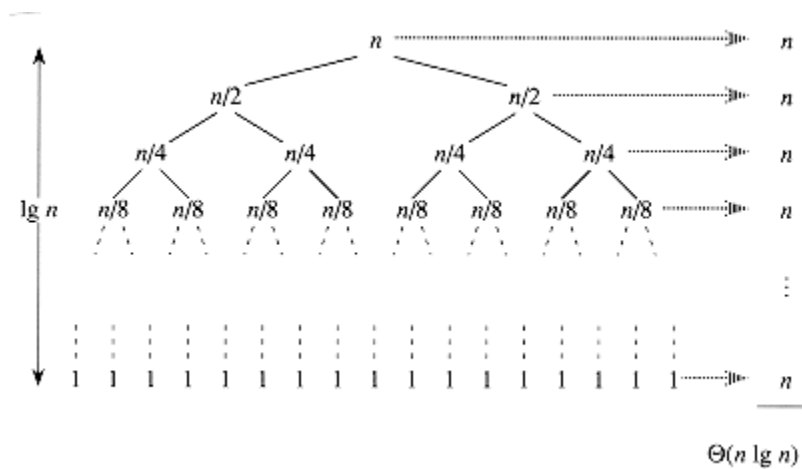
Complejidad espacial Quicksort incluye variables de metodos auxiliares

	Complejidad espacial	
	variable	cantidad de valores atomicos
Entrada	start	1
	end	1
	array	n
salida	resultado	n
auxiliares	start	1
	middle	1
	end	1
	i	1
	j	1
	position	1
	list aux	n
resultado		3n+8

Complejidad temporal del quicksort

Algoritmo	#Codigo	# que se repite cada linea
Quicksort	quickSortR(array,0,array.size()-1,comparator);	1
QuicksortR	int firstStart = start;	1
	int firstEnd = end;	1
	if(end-start+1<=2)	n
	if(end<array.size() && start<array.size() && end >start)	n
	if(comparator.compare(array.get(start), array.get(end))>0)	n
	exchange(array,start,end);	3
	else	
	int pivot= (start+end)/2;	1
	boolean order = false;	n
	boolean alreadyOrdained = true;	1
	while(!order)	Log n+1
	boolean lower = false;	1
	boolean upper = false;	1
	if(comparator.compare(array.get(start), array.get(pivot))<0)	n
	start++;	n
	else if(start<=pivot)	n
	lower = true;	1
	if(comparator.compare(array.get(end), array.get(pivot))>0)	n
	end--;	n
	else if(end>=pivot)	n
	upper = true;	1
	if(lower&&upper)	n

	if(pivot == end)	n
	pivot = start;	1
	else if(pivot == start)	n
	pivot = end;	1
	alreadyOrdained = false;	1
	exchange(array,end,start);	3
	if(start<pivot)	n
	start++;	n
	if(end>pivot)	n
	end--;	n
	order = end == start && end == pivot;	n
	if(!alreadyOrdained)	n
	quickSortR(array,firstStart,pivot-1,comparator);	Log n
	quickSortR(array,pivot+1,firstEnd,comparator);	Log n
TOTAL		$O(n \log n)$



```

58
59 public static <T> void mergeSort(List<T> array, Comparator<T> comparator){
60     mergeSortR(array,0,array.size()-1,comparator);
61 }
62 public static <T> void mergeSortR(List<T> array, int start,int end,Comparator<T> comparator) {
63     if(end-start+1>=2) {
64         mergeSortR(array,start,(start+end)/2,comparator);
65         mergeSortR(array,((start+end)/2)+1,end,comparator);
66     }
67     toCombine(array,start,(start+end)/2,end,comparator);
68 }
69 private static <T> void toCombine(List<T>array, int start, int middle, int end,Comparator<T> comparator) {
70     int i = start;
71     int j = middle +1;
72     List<T> listAux = new ArrayList<T>();
73     if((end-start+1)>1) {
74         while(i<=middle && j<=end) {
75
76             if(comparator.compare(array.get(i), array.get(j))<0) {
77                 listAux.add(array.get(i));
78                 i++;
79             }else {
80                 listAux.add(array.get(j));
81                 j++;
82             }
83         }
84         for (int k = i; k <= middle; k++) {
85             listAux.add(array.get(k));
86         }
87         for (int k = j; k <= end; k++) {
88             listAux.add(array.get(k));
89         }
90     }
91     int position = start;
92     for (int k = 0; k < listAux.size(); k++) {
93
94         array.set(position,listAux.get(k));
95         position++;
96     }
97 }
98 }

```

Complejidad espacial Merge Sort

	Complejidad espacial	
	variable	cantidad de valores atomicos
Entrada	start	1
	end	1
	array	n
salida	resultado	n
	start	1
	middle	1
auxiliares	end	1
	i	1
	j	1
resultado		2n+8

Complejidad temporal merge sort

Algoritmo	Linea de codigo	# veces que se repite
Mergesort	mergeSortR(array,0,array.size()-1,comparator);	1
MergeSortR	if(end-start+1>=2)	n
	mergeSortR(array,start,(start+end)/2,comparator);	n
	mergeSortR(array,((start+end)/2)+1,end,comparator);	n
	toCombine(array,start,(start+end)/2,end,comparator);	$2x^2+10+9$
To combine	int i = start;	1
	int j = middle +1;	1
	List<T> listAux = new ArrayList<T>();	1
	if((end-start+1)>1)	n
	while(i<=middle && j<=end)	n+1
	if(comparator.compare(array.get(i), array.get(j))<0)	n(n+1)
	listAux.add(array.get(i));	n
	i++;	n
	else	
	listAux.add(array.get(j))	n
	j++;	n
	for (int k = i; k <= middle; k++)	n(n+1)
	listAux.add(array.get(k))	1
	for (int k = j; k <= end; k++)	n
	listAux.add(array.get(k))	1
	int position = start;	1
	for (int k = 0; k < listAux.size(); k++)	n+1
	array.set(position,listAux.get(k));	1
	position++;	n
TOTAL		$2x^2+10+9 = N (n \log n)$

