PREDATOR:
Justificación de las operaciones.

1.La clase Area.

1.1Operación *lucha*:

La operación lucha hace que todas las especies que hay en el área luchen por su supervivencia, ya sea devorando a otras especies (carnívoros) o alimentándose de la vegetación de la zona (hervívoros). Los carnívoros tienen unas calorias mínimas que ingerir para sobrevivir; van alimentándose de las demás especies según sus preferencias alimenticias que establecen un orden (solo atacan a las especies que se encuentren entre sus preferentes). Si no hay indivíduos que ingerir para poder cumplir con el mínimo de calorías, el especimen muere.

Por otro lado los hervívoros no se preocupan de nada, y si no son deborados, no mueren.

1.1.1 Implementación:

```
//pre: cierto
void Area::lucha_region(Cjt_especies e) {
	for (int i = 0; i < e.num_carnivoros(); ++i) {
		int n = e.turno_iesimo(i);
		int k = especies[n-1];
		Especie o = e.consultar_esp(n);
		int it = 0;
		while (k > 0) {
			int cal = o.calorias();
			while (cal > 0 and it < o.npres()) {
				int j = o.presa_iesima(it);
				if (especies[j - 1] > 0) {
					Especie g = e.consultar_esp(j);
					--especies[j - 1];
					cal = cal - g.masa();

				}
				else ++it;
			}
			if (cal > 0) --especies[n - 1];
			--k;
		}
	}
} //post: se ha realizado la lucha en el área.
```

1.1.2 Justificación:

Tenemos que justificar la operación de alimentación de una sola especie, así pues entramos en la función lucha_region() hasta while(k >0), donde comienza a tratar uno a uno los individuos de la especie.

En esta función hay barios invariantes: el primero 0 <= k <= nº de indivíduos de esa especie en el área, el segundo 0 <= cal <= calorias mínimas que necesita la especie, y el tercero es que 0 <= it <= nº de presas de la especie a tratar.

Inicialmente no hemos tratado ningún elemento, por lo tanto it = 0 para que comiencen atacando a su primera presa, cumpliendo así el tercer invariante. K toma el valor del número de indivíduos de la especie que estamos tratando, pues todos tienen que actuar (bien comiendo y sobreviviendo o bien muriendo); así hacemos que se cumpla el primer invariante.
Como el primer "animal" aun no ha atacado a ninguna presa inicializamos calorias con su valor máximo para ir restando a este valor el de las calorías que le proporcione cada presa, manteniendolo en el rango que marca el invariante dos, así que este también se cumple.

Todas las operaciones auxiliares a las que llama (consultar_esp(), calorias(), npres() y masa()) tienen como precondicion cierto, por tanto no hay problemas en las llamadas a excepción de presa_iesima(int i), que tiene por precondición que la i esté en el rango [0, numero de presas]. También en este caso todas las llamadas son válidas pues la condición de salida del while donde se utiliza la función es que el parámetro que le pasamos no se pase de este rango.

Una vez dendtro del bucle (while (k > 0)) solo se puede salir si k <= 0, esto indicaría que todos los indivíduos de la especie en este área han luchado, por tanto se cumpliría la postcondición. Sabemos que k llegara a valer 0 por la operación –k que hay al final del bucle.
Dentro del segundo bucle (while (cal.....)) hay dos maneras de salir del mismo, una es que cal <= 0, lo cual indica que el animal ha ingerido suficientes calorias para sobrevivir y no necesita comer más, por tanto cede el turno al siguiente. Sabemos que llegara este momento pues cada vez que encuentra una presa para comer se le resta a cal las calorias que le ha aportado su presa (cal = cal – g.masa()). La segunda forma de salir de este bucle es hacer que it == nº de presas, esto lo conseguimos en caso de que un depredador se coma a su última presa, pues incrementaremos it para buscar su siguiente especie presa, pero al no haber más salremos del bucle.

Visto todo esto queda demostrado que el programa terminará y lo hará cumpliendo los invariantes y la postcondición.

2.La clase Cjt_areas.

2.1.Operación *migracion_b*:

Esta operación de la clase Cjt_areas “simula” la migración de una especie indicada, desde un área indicada, en dirección hacia la planicie. Nos permite decidir cuantos indivíduos de la especie dejan el área actual, y también introducir un factor g que será determinante a la hora de realizar la migración.

2.1.1 Implementación:

```
//pre: cierto
pair<bool, int> Cjt_areas::migracion_b(Arbre<int> a, int r, int e, int h, int g) {
	pair<bool, int> b;
	if (a.arrel() != r) {
		Arbre<int> a1;
		Arbre<int> a2;
		int raiz = a.arrel();
		a.fills(a1, a2);
		if (not a1.es_buit()) {
			b = migracion_b(a1, r, e, h, g);
			if (b.first)  {
				if (es_planicie(raiz)) {
					areas[raiz - 1].nacimiento(e, b.second);
				}
				else {
					if (b.second >= g) {
						areas[raiz - 1].nacimiento(e, b.second/2);
						b.second -= b.second/2;
						b.first = true;
					}
					else {
						areas[raiz - 1].nacimiento(e, b.second);
						b.first = false;
						b.second = 0;
					}
				}
			}
			else {
				if (not a2.es_buit()) {
					b = migracion_b(a2, r, e, h, g);
					if (b.first)  {
						if (es_planicie(raiz)) {
							areas[raiz - 1].nacimiento(e, b.second);
						}
						else {
							if (h >= g) {
								areas[raiz - 1].nacimiento(e, b.second/2);
								b.second -= b.second/2;
```

```
                                                b.first = true;
                                        }
                                        else {
                                                areas[raiz - 1].nacimiento(e, b.second);
                                                b.first = false;
                                                b.second = 0;
                                        }
                                }
                        }
                }
        }

        else {
                areas[r - 1].nacimiento(e, h*(-1));
                b.first = true;
                b.second = h;
        }
        return b;
}

//post: se ha realizado la migración hacia la planicie
```

2.1.2 Justificación:

El caso base (que el área a la que accedemos es r) decrementa en h el número de indivíduos de la especie e en ese área, de forma que “simula” que han abandonado el área. Por tanto en caso de tener un arbol con dos únicos nodos, se cumpliría la postcondición por parte del caso base.

Ahora se puede justificar el caso recursivo sabiendo que el caso base es correcto. Lo primero que hace es comprovar si el arbol tiene hijo izquierdo, pues si no lo tiene significa que es un área periférica (pues según el enunciado si no existe hijo izquierdo tampoco existe el derecho), con lo cual no hay que hacermás llamadas recursivas. En el caso de tener hijo izquierdo hace una llamada a migracion_b pasando por parámetro su hijo izquierdo para saber si en el se encuentra el área origen de la migración (en cullo caso b.fist = true), y en ese caso cuantos especímenes llegan hasta este punto (b.second). Si el hijo izquierdo no contiene r, entonces comprueba si dicha región se encuentra en su hijo derecho.

Una vez se conoce qué hijo contiene ese área, se comprueba si el área actual es la planicie o no, pues si lo fuera no hace falta tener en cuenta el factor g y todos los especímenes de la especie e que han llegado hasta aquí, se quedan. En caso de no estar en la planicie comprovamos el factor g para saber los especímenes que se quedarán en el área y los que migrarán al área superior, pues este número se tiene que indicar en b.second para que el padre del área actual que ha realizado la llamada a migracion_b sepa cuantos especímenes llegan a el.

Ahora vamos a comprovar que todas las llamadas a funciones sean correctas, lo haré desde arriba hacia abajo.

La primera llamada es a a.arrel(), cuya precondición es que a no esté vacío, cosa que podemos asegurar, pues cuando el programa principal llama a migración, siempre le pasará un arbol de regiones con un mínimo de nodos mayor que 0, pues sino no tendría sentido; y en el caso de que sea migracion_b la que la llama, antes de realizar dicha llamada nos aseguramos con if que el arbol que pasamos no esté vacío.

a.fills() tiene como precondición que a no esté vacío (demostrado en el párrafo anterior que no lo está) y que a1 y a2 si lo estén, y lo están pues se declaran justo antes de la llamada.

La siguiente es a.buit(), que tiene precondicion cierta, por tanto se cumple siempre. Le sigue migracion_b, que por hipótesis de inducción suponemos correcta. Después biene b.first, consultora de la clase pair que también tiene por precondición cierto.

Ahora llamamos a es_planicie(), operación de Cjt_areas que también tiene por precondición cierto. La siguiente que toca mencionar es nacimiento, se la clase Area, que también tiene precondición cierto.

La última por comprovar es b.second (operación consultora de la clase pair), que también tiene precondición cierto, y por tanto, se cumplirá siempre.

La finalización del programa se basa en encontrar r o en llegar a un área periférica. En el primer caso sebolveremos b, con b.first igual a true, para indicar que en esa rama del arbol se encuentra r y que van a llegar indivíduos al área superior (la que ha realizado la llamada), a partir de ese momento no se realizan más llamadas recursivas de la función y el programa termina. En el otro caso, que nos encontremos en una región periférica, ocurre lo mismo, estamos al final del arbol y comienza el camino de retorno hasta el primer nodo (la planicie) que ha realizado la primera llamada.