

**KAUNO TECHNOLOGIJOS UNIVERSITETAS**

**INFORMATIKOS FAKULTETAS**

**DUOMENŲ STRUKTŪROS**

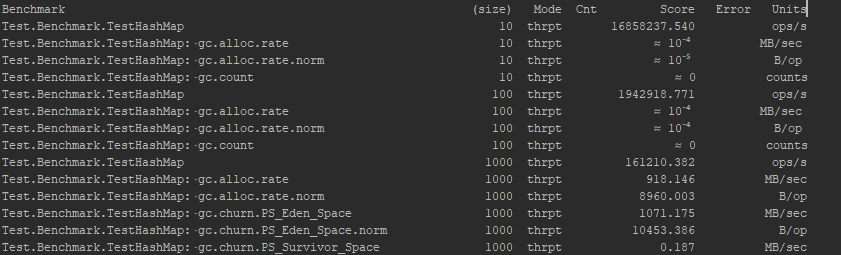
**PROJEKTINIS DARBAS**

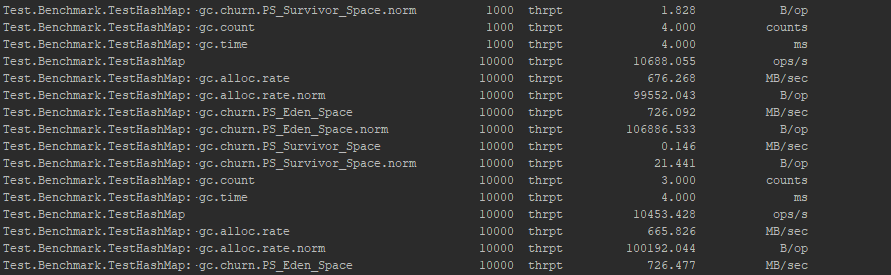
**Atliko Karolis Stasaitis IFF8-9**

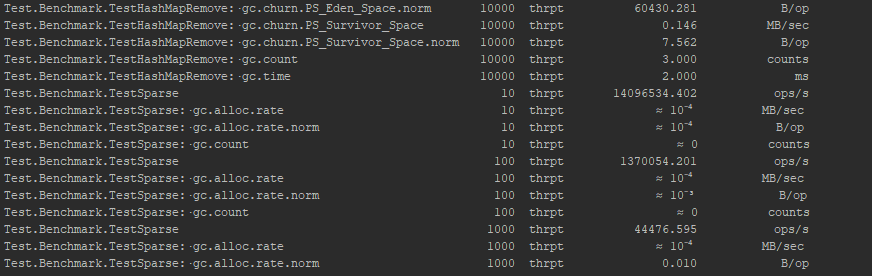
**Pagrindinė Klasė:**

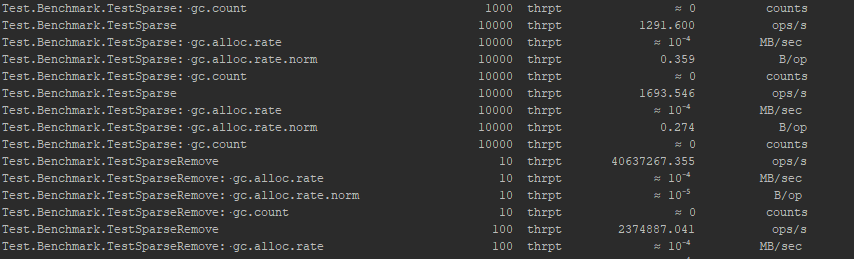
**package** Test;  
  
**public class** SparseArray<E> **implements** Cloneable {  
 **private static final** Object ***DELETED*** = **new** Object();  
 **private int**[] **mKeys**;  
 **private** Object[] **mValues**;  
 **private int mSize**;  
  
 **public** SparseArray()  
 {  
 **this**(10);  
 }  
  
 **public** SparseArray(**int** initialCapacity) {  
 **if** (initialCapacity == 0) {  
 **mKeys** = **new int**[0];  
 **mValues** = **new** Object[0];  
 }  
 **else** {  
  
 **mValues** = **new** Object[initialCapacity];  
 **mKeys** = **new int**[**mValues**.**length**];  
 }  
 **mSize** = 0;  
 }  
 **public** SparseArray<E> clone() **throws** CloneNotSupportedException{  
 SparseArray<E> clone = **null**;  
 clone = (SparseArray<E>) **super**.clone();  
 clone.**mKeys** = **mKeys**.clone();  
 clone.**mValues** = **mValues**.clone();  
 **return** clone;  
 }  
 **public** E get(**int** key) {  
 **return** get(key, **null**);  
 }  
 **public** E get(**int** key, E valueIfKeyNotFound) {  
 **int** i = binarySearch(**mKeys**,**mSize**,key);  
 **if** (i < 0 || **mValues**[i] == ***DELETED***) {  
 **return** valueIfKeyNotFound;  
 } **else** {  
 **return** (E) **mValues**[i];  
 }  
 }  
 **public int** size() {  
 **return mSize**;  
 }  
 **public int** indexOfValue(E value) {  
 **for** (**int** i = 0; i < **mSize**; i++) {  
 **if** (**mValues**[i] == value) {  
 **return** i;  
 }  
 }  
 **return** -1;  
 }  
 **public int** indexOfKey(**int** key) {  
 **return** binarySearch(**mKeys**, **mSize**, key);  
 }  
 **public void** clear() {  
 **int** n = **mSize**;  
 Object[] values = **mValues**;  
 **for** (**int** i = 0; i < n; i++) {  
 values[i] = **null**;  
 }  
 **mSize** = 0;  
 }  
 **public void** append(**int** key, E value) {  
 **if** (**mSize** != 0 && key <= **mKeys**[**mSize** - 1]) {  
 put(key, value);  
 **return**;  
 }  
 **mKeys** = append(**mKeys**, **mSize**, key);  
 **mValues** = append(**mValues**, **mSize**, value);  
 **mSize**++;  
 }  
 **public void** setValueAt(**int** index, E value) {  
 **if** (index >= **mSize**) {  
 **throw new** ArrayIndexOutOfBoundsException(index);  
 }  
 **mValues**[index] = value;  
 }  
 **public void** put(**int** key, E value) {  
 **int** i = binarySearch(**mKeys**, **mSize**, key);  
 **if** (i >= 0) {  
 **mValues**[i] = value;  
 }  
 **else** {  
 i = ~i;  
 **if** (i < **mSize** && **mValues**[i] == ***DELETED***) {  
 **mKeys**[i] = key;  
 **mValues**[i] = value;  
 **return**;  
 }  
 **mKeys** = *insert*(**mKeys**, **mSize**, i, key);  
 **mValues** = *insert*(**mValues**, **mSize**, i, value);  
 **mSize**++;  
 }  
 }  
 **public int** keyAt(**int** index) {  
 **if** (index >= **mSize**) {  
 **throw new** ArrayIndexOutOfBoundsException(index);  
 }  
 **return mKeys**[index];  
 }  
 **public** E valueAt(**int** index) {  
 **if** (index >= **mSize**) {  
 **throw new** ArrayIndexOutOfBoundsException(index);  
 }  
 **return** (E) **mValues**[index];  
 }  
 **public void** remove(**int** key) {  
 **int** i = binarySearch(**mKeys**, **mSize**, key);  
 **if** (i >= 0) {  
 **if** (**mValues**[i] != ***DELETED***) {  
 **mValues**[i] = ***DELETED***;  
 }  
 }  
 }  
 **public void** removeAt(**int** index) {  
 **if** (index >= **mSize**) {  
 **throw new** ArrayIndexOutOfBoundsException(index);  
 }  
 **if** (**mValues**[index] != ***DELETED***) {  
 **mValues**[index] = ***DELETED***;  
 }  
 }  
 **public** String toString() {  
 **if** (size() <= 0) {  
 **return "{}"**;  
 }  
 StringBuilder buffer = **new** StringBuilder(**mSize** \* 28);  
 buffer.append(**'{'**);  
 **for** (**int** i=0; i<**mSize**; i++) {  
 **if** (i > 0) {  
 buffer.append(**", "**);  
 }  
 **int** key = keyAt(i);  
 buffer.append(key);  
 buffer.append(**'='**);  
 Object value = valueAt(i);  
 **if** (value != **this**) {  
 buffer.append(value);  
 } **else** {  
 buffer.append(**"(this Map)"**);  
 }  
 }  
 buffer.append(**'}'**);  
 **return** buffer.toString();  
 }  
 **private** <T> T[] append(T[] keys, **int** size, T value)  
 {  
 **if**(size + 1 > keys.**length**){  
 T[] array = (T[]) **new** Object[size];  
 System.*arraycopy*(array, 0, array, 0,size);  
 keys = array;  
 }  
 keys[size] = value;  
 **return** keys;  
 }  
 **private int**[] append(**int**[] keys, **int** size, **int** value)  
 {  
 **if**(size + 1 > keys.**length**){  
 **int**[] array = **new int**[size];  
 System.*arraycopy*(array, 0, array, 0,size);  
 keys = array;  
 }  
 keys[size] = value;  
 **return** keys;  
 }  
 **private static int**[] insert(**int**[] array, **int** size, **int** index, **int** element)  
 {  
 **if** (size + 1 <= array.**length**) {  
 System.*arraycopy*(array, index, array, index + 1, size - index);  
 array[index] = element;  
 **return** array;  
 }  
 **int**[] newa = **new int**[*growSize*(size)];  
 System.*arraycopy*(array, 0, newa, 0, index);  
 newa[index] = element;  
 System.*arraycopy*(array, index, newa, index + 1, array.**length** - index);  
 **return** newa;  
 }  
 **private static** <T> T[] insert(T[] array, **int** size, **int** index, T element)  
 {  
 **if** (size + 1 <= array.**length**) {  
 System.*arraycopy*(array, index, array, index + 1, size- index);  
 array[index] = element;  
 **return** array;  
 }  
 T[] newa = (T[]) **new** Object[*growSize*(size)];  
 System.*arraycopy*(array, 0, newa, 0, index);  
 newa[index] = element;  
 System.*arraycopy*(array, index, newa, index + 1, array.**length** - index);  
 **return** newa;  
 }  
 **private static int** growSize(**int** currentSize) {  
 **return** currentSize <= 4 ? 8 : currentSize \* 2;  
 }  
 **private int** binarySearch(**int**[] array, **int** size, **int** value)  
 {  
 **int** lo = 0;  
 **int** hi = size - 1;  
 **while** (lo <= hi) {  
 **final int** mid = (lo + hi) >>> 1;  
 **final int** midVal = array[mid];  
 **if** (midVal < value) {  
 lo = mid + 1;  
 } **else if** (midVal > value) {  
 hi = mid - 1;  
 } **else** {  
 **return** mid;  
 }  
 }  
 **return** ~lo;  
 }  
 **public void** Iterate()  
 {  
 **for** (**int** i = 0; i < **mSize**; i++) {  
  
 **int** key = keyAt(i);  
 String value = valueAt(i).toString();  
  
 System.***out***.println(**"key: "** + key + **"value:"** + value);  
 }  
 }  
}

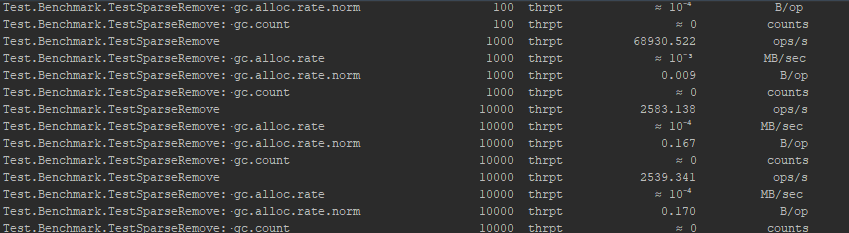
**Benchmark.java**











**Žėdami į –gc.allor-rate-norm matome, kad SparseArray reikalauja mažiau memory ant šiukšliu surinkimo tai galime teigti, kad SparseArray naudoja mažiau atminties.**

**Greitaveikos rezultatai:**

**Išvados**

SparseArray yra naudojamas android systemose, nes Sparsearray sutaupo memory allocation sanaudas.Sužinojau daugiau apie JMH, nes norėjau pažėti ar tikrai sutaupo atminties ir optimizuotai.