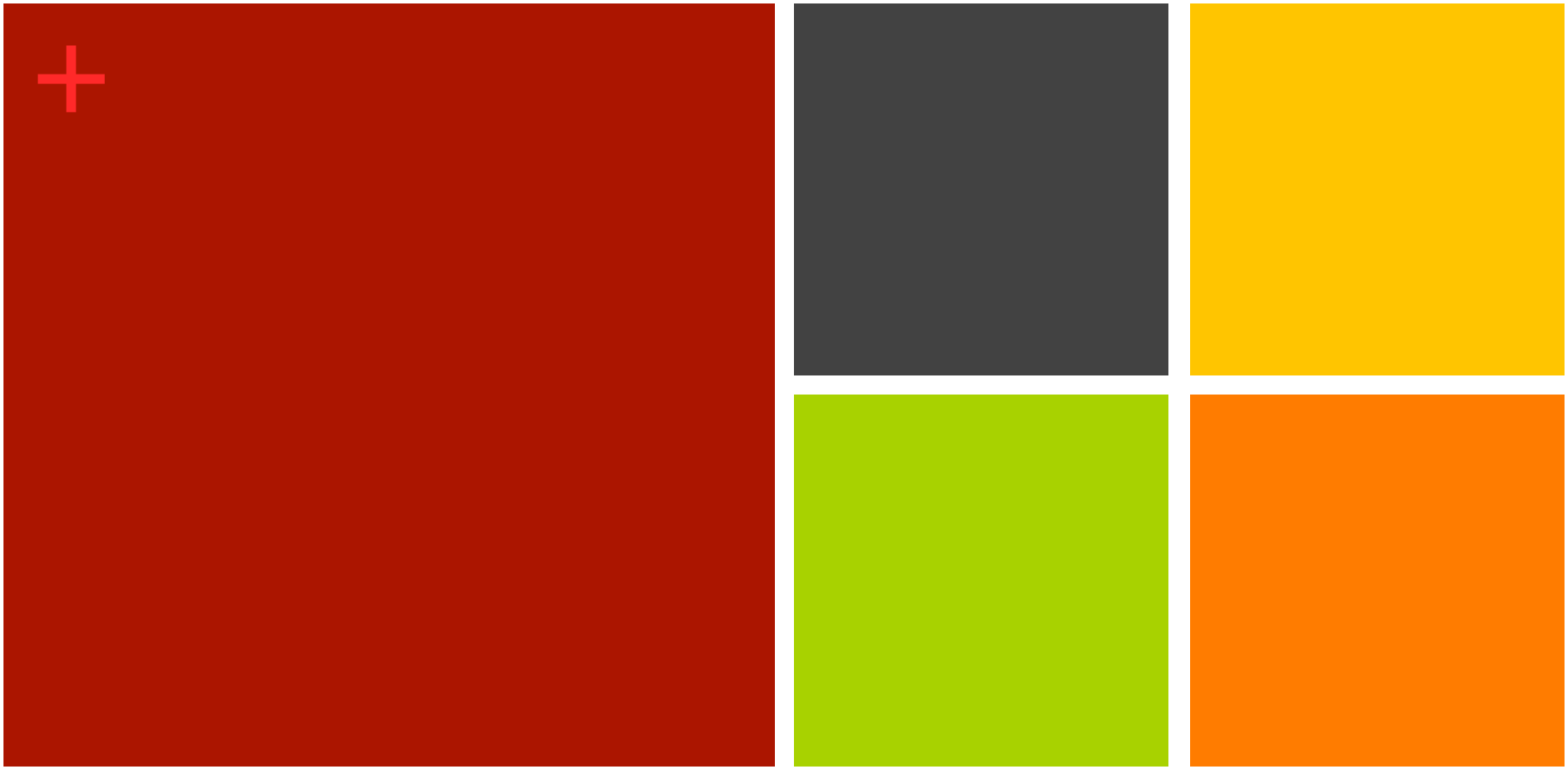


Bloom Filters

Filippo Todeschini
Laboratory on Algorithms for Big Data
Rossano Venturini

Bloom Filter: Introduzione

- Struttura dati probabilistica
- Rappresentazione succinta di un insieme
- Rispondere ad operazioni di appartenenza di un elemento nell'insieme
- Falsi Positivi
- Applicazioni principali: interrogazione database, condivisione e trasmissione di liste
- Implementazione semplice e personalizzabile

Bloom Filter: Problema

- Dato un insieme $S=\{x_1, x_2, \ldots, x_n\}$ di n elementi, rappresentare tale insieme in un vettore di m bit utilizzando k funzioni hash h().
- Le k funzioni hash h() devono essere indipendenti tra loro e devono avere tutte lo stesso range di valori [0, …, m-1]
- La probabilità di ottenere un FP

$$fp = (1 - (1 - \frac{1}{m})^{kn})^k = (1 - e^{-k\frac{n}{m}})^k$$

- Valori ottimali di k e m dipendono da n e dal valore di FP desiderata

$$m = \frac{-\ln p}{\ln^2 2} n \qquad\qquad k = \frac{m}{n} \ln 2$$

- Garantisce le operazioni di lookup() e add() in tempo O(k)

Funzioni Hash

- K funzioni hash indipendenti con un range [0,…, m-1]
- Double hashing $$h(i,k) = (h_1(k) + i * h_2(k)) mod |T|$$
- Murmur Hash

```
1  public static int getHashing(byte[] element, int i, int m, String
       type){
2    int hash=(hash(element,type,seedhash1)+ hash(element,type,
         seedhash2) * i) % m;
3    if( hash < 0 ) hash += m;
4    return hash;
5  }
6  private static int hash( byte[] key, String type, int seed){
7    switch(type){
8      case "Murmur3":return MurmurHash3.murmurhash3_x86_32(key, 0,
           key.length, seed);
9      case "Murmur2":return MurmurHash2.hash32(key, key.length, seed)
           ;
10     case "Murmur":return MurmurHash.hash(key, seed);
11   }
12 }
```

Standard Bloom Filter (SBF)

- Inizializzazione

```
StandardBloomFilter sbf=new StandardBloomFilter(0.001, 10000,"
    Murmur3");
```

```
private boolean initialize(double aFP,int expectedNElement, String
    typeHashing){
    n=expectedNElement;
    fp_obj=aFP;
    m=(int)Math.round(-n * Math.log(fp_obj)/Math.pow(Math.log(2),
        2));
    k=(int)Math.round((double)m/(double)n* Math.log(2));
    this.typeHashing=typeHashing;
    this.set= new BitSet(m);
    return true;
 }
```

- FP probability

```
public double getProbabilityFP(){
    return Math.pow((1 - Math.exp(-k * (double) n / (double) m)), k
        );
  }
public double getProbabilityFPReal(){
    return Math.pow((1 - Math.exp(-k * (double)
        numberOfAddedElement/ (double) m)), k);
  }
```

- Spazio occupato O(m)

Add(x)

```
private void addLocal(byte[] element){
  for(int j=0;j<k;j++){
    set.set(BFHashing.getHashing(element, j, m, this.
        typeHashing), true);
  }
  numberOfAddedElement++;
}
```

Costo: O(k)

Lookup(x)

```
public boolean lookup(byte[] element){
  for(int j=0;j<k;j++)
  if(set.get(BFHashing.getHashing(element, j, m, this.
      typeHashing))==false)return false;
  return true;
}
```

Costo: O(k)

Counting Bloom Filter (CBF)

Utilizza un vettore di contatori invece che un vettore di bit; ciò comporta un aumento dello spazio utilizzato, ma permette l'operazione di delete().

- Inizializzazione

```
1  private void initialize(double aFP,int expectedNElement,String
       typeHashing,Class<?> cl){
2    n=expectedNElement;
3    fp_obj=aFP;
4    m=(int)Math.round(-n * Math.log(fp_obj)/Math.pow(Math.log(2), 2))
         ;
5    k=(int)Math.round((double)m/(double)n* Math.log(2));
6    this.typeHashing=typeHashing;
7    switch(cl.getSimpleName()){
8      case "Byte":{setByte=new byte[m];setShort=null;setInt=null;
           break;}
9      case "Short":{setShort=new short[m];setByte=null;setInt=null;
           break;}
10     case "Integer":{setInt=new int[m];setShort=null;setByte=null;
           break;}
11   }
12   cl=null;
13 }
```

- FP probability
- Spazio occupato O(m)

Add(x)

```
1  private void addLocalInt(byte[] element){
2      for(int j=0;j<k;j++){
3        setInt[BFHashing.getHashing(element, j, m, this.
             typeHashing)]=(int)setInt[BFHashing.getHashing(
             element, j, m, this.typeHashing)]+1;
4      }
5      numberOfAddedElement++;
6    }
```

Costo: O(k)

Lookup(x)

Costo: O(k)

Delete(x)

```
1  private void deleteLocalInt(byte[] element){
2      for(int j=0;j<k;j++){
3        setInt[BFHashing.getHashing(element, j, m, this.
             typeHashing)]=(int)setInt[BFHashing.getHashing(
             element, j, m, this.typeHashing)]-1;
4      }
5      numberOfAddedElement--;
6    }
```

Costo: O(k)

Compressed Bloom Filter (CBF)

- Ridurre lo spazio utilizzato dai Bloom Filter, mantenedo però la FP inalterata
- Utilizzati nella condivisione e trasmissione dei Bloom Filter nella rete

- Considerando uno SBF con un vettore di m bit e k funzioni hash, il CBF è costruito a partire da valori :
  - $z = m * \ln 2$
  - $k = z/n * \ln 2$
- Il numero k di funzioni hash è maggiore di quello del SBF iniziale.
- Spazio occupato O(z), tale valore permette di risparmiare circa il 30% dello spazio.

Inizializzazione

```
1  private void initialize(double aFP,int expectedNElement, String
       typeHashing){
2    n=expectedNElement;
3    fp_obj=aFP;
4    m=(int)Math.round(-n * Math.log(fp_obj)/Math.pow(Math.log(2), 2))
         ;
5    m = z = (int) Math.round(m*Math.log(2));
6    k=(int)Math.round((double)-z/(double)n * Math.log(fp_obj));
7    this.typeHashing=typeHashing;
8    set=new BitSet(m);
9    }
```

Add(x)

Costo: O(k)

Lookup(x)

Costo: O(k)

Delete(x)

Costo: O(k)

Dynamic Bloom Filter

Variante dinamica dei Bloom Filter: questa permette di rappresentare un insieme dinamico di elementi, mantenendo un FP molto vicino a quello desiderato.

- L'idea è quella di utilizzare un insieme di SBF tutti con le stesse caratteristiche.
- Ogni qual volta un SBF è saturo se ne aggiunge un altro.

- FP probability

$$fp = \begin{cases} 1-(1-e^{-k\frac{n}{m}})^k & s=1 \\ 1-(1-(1-e^{-k\frac{n}{m}})^k)^{\lfloor \frac{nAddedElement}{c} \rfloor}(1-(1-e^{-k\frac{nAddedElement-c\lfloor \frac{nAddedElement}{c} \rfloor}{m}})^k) & s>1 \end{cases}$$

  - dove
    - s è il numero di SBF
    - c è la capacità dei singoli SBF
- Spazio occupato O(s * m)

Add(x)

```
1  private void addLocal(byte[] element){
2    if(bfList.get(this.indexActiveBF).isFull()==false){
3      bfList.get(this.indexActiveBF).add(element);
4      numberOfAddedElement++;
5    }else{
6      boolean b=addBloomFilter();
7      addLocal(element);
8    }
9  }
10 private boolean addBloomFilter(){
11   bfList.add(new StandardBloomFilter(fp_obj,c));
12   indexActiveBF=bfList.size()-1;
13   return true;
14 }
```

Costo: O(k)

Lookup(x)

```
1  public boolean lookup(byte[] element){
2    for(int j=0;j<this.bfList.size();j++){
3      if(bfList.get(j).lookup(element)==true)return true;
4    }
5    return false;
6  }
```

Costo: O(s * k)

Spectral Bloom Filter (SPBF)

Variante del Counting Bloom Filter che consente una nuova operazione: frequency(x).

Frequency (x)

La frequenza di un elemento è data dal minimo valore tra i vari contatori relativi alle h_k funzioni hash.

Minimum Insertion

- Operazioni consentite
  - add()
  - lookup()
  - frequency()
- Invece di incrementare tutti i contatori relativi alle h_k funzioni hash, si incrementano solo quelli il cui valore è il minimo.

Recurring Minimum

- Operazioni consentite
  - add()
  - lookup()
  - frequency()
  - delete()
- Invece di incrementare tutti i contatori relativi alle h_k funzioni hash, si incrementano solo quelli il cui valore è il minimo.
- Utilizza un secondo vettore per salvare gli elementi che hanno un minimo unico.
- Questo secondo vettore contiene al massimo il 20% degli elementi attesi n

Spectral Bloom Filter (SPBF)
Minimum Insertion

Minimum Insertion

- Operazioni consentite
  - add()
  - lookup()
  - frequency()
- Invece di incrementare tutti i contatori relativi alle h_k funzioni hash, si incrementano solo quelli il cui valore è il minimo.

Lookup(x)

Costo: O(k)

Add(x)

```
1  private void addLocalInt(byte[] element){
2    HashMap<Integer,ArrayList<Integer>> map=new HashMap<Integer,
         ArrayList<Integer>>();
3    for(int j=0;j<k;j++){
4      int key=BFHashing.getHashing(element, j, m, typeHashing);
5      ArrayList<Integer> a= map.get(setInt[key]);
6      try{
7        a.add(key);
8      }catch(Exception e){
9        a=new ArrayList<Integer>();
10       a.add(key);
11     }
12     map.put(setInt[key], a);
13   }
14   int min= Collections.min(map.keySet());
15   ArrayList<Integer> indexList=map.get(min);
16   for(int j : indexList) setInt[j]=(int)(min+1);
17   numberOfAddedElement++;
18   }
```

Costo: O(k)

Frequency(x)

```
1  private int frequencyLocalInt(byte[] element){
2    int minValue=0;
3    for(int j=0;j<k;j++){
4      int hash=setInt[BFHashing.getHashing(element, j, m, this.
           typeHashing)];
5      if(hash==0) return 0;
6      else if((minValue>hash || j==0 ) && hash!=0)minValue=hash;
7    }
8    return minValue;
```

Costo: O(k)

Spectral Bloom Filter (SPBF) Recurring Minimum

Add(x)

```
1  private void addLocalInt(byte[] element){
2    HashMap<Integer,ArrayList<Integer>> map=new HashMap<Integer,
         ArrayList<Integer>>();
3    for(int j=0;j<k;j++){
4      int key=BFHashing.getHashing(element, j, m, typeHashing);
5      setInt[key]=setInt[key]+1;
6      ArrayList<Integer> a= map.get(setInt[key]);
7      try{
8        a.add(key);
9      }catch(Exception e){
10       a=new ArrayList<Integer>();
11       a.add(key);
12     }
13     map.put(setInt[key], a);
14   }
15   int min= Collections.min(map.keySet());
16   ArrayList<Integer> indexList=map.get(min);
17   if(indexList.size()==1){
18     int increment=min;
19     for(int j=0;j<k2;j++){
20       if(setInt2[BFHashing.getHashing(element, j, m2,
             typeHashing)]==0){
21         increment=1;
22         break;
23       }
24     }
25     for(int j=0;j<k2;j++){
26       int key=BFHashing.getHashing(element, j, m2, typeHashing
             );
27       setInt2[key]=(setInt2[key]+increment);
28     }
29     uniqueItems++;
30   }
31   numberOfAddedElement++;
32 }
```

Costo: O(k)

Lookup(x)

Costo: O(k)

Delete(x)

Costo: O(k)

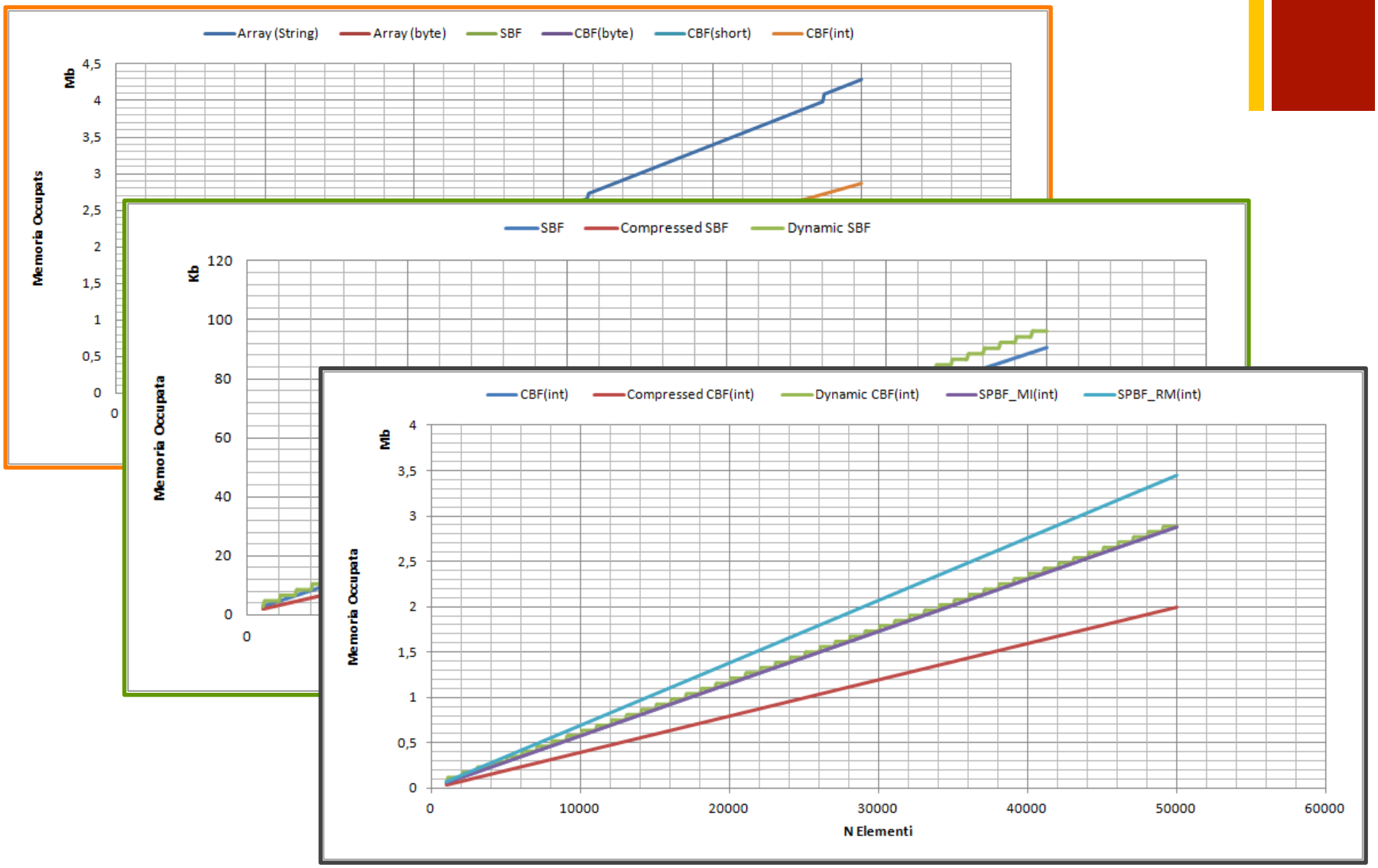
Metodologia analoga all'operazione di add(), solamente che decrementa invece che incrementare

Frequency(x)

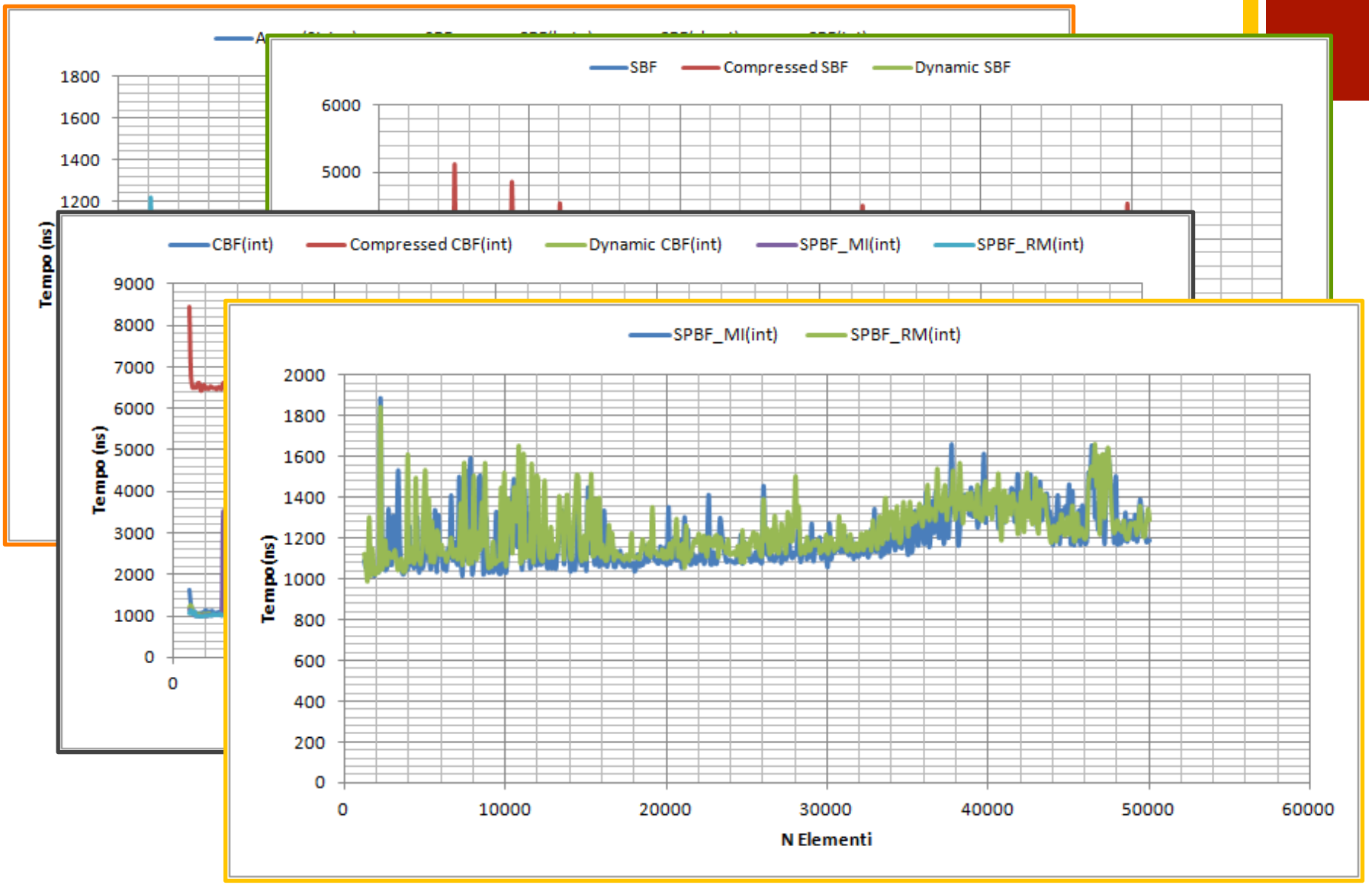
```
1  private int frequencyLocalInt(byte[] element){
2    HashMap<Integer,ArrayList<Integer>> map=new HashMap<Integer,
         ArrayList<Integer>>();
3    for(int j=0;j<k;j++){
4      int key=BFHashing.getHashing(element, j, m, typeHashing);
5      if(setInt[key]==0) return 0;
6      ArrayList<Integer> a= map.get(setInt[key]);
7      try{
8        a.add(key);
9      }catch(Exception e){
10       a=new ArrayList<Integer>();
11       a.add(key);
12     }
13     map.put(setInt[key], a);
14   }
15   int min= Collections.min(map.keySet());
16   ArrayList<Integer> indexList=map.get(min);
17   if(indexList.size()==1){
18     int min2=0;
19     for(int j=0;j<k2;j++){
20       int val=setInt2[BFHashing.getHashing(element, j, m2,
             typeHashing)];
21       if(val==0) return min;
22       else if(val<min2 || j==0) min2=val;
23     }
24     return min2;
25   }else{
26     return min;
27   }
28 }
```

Costo: O(k)

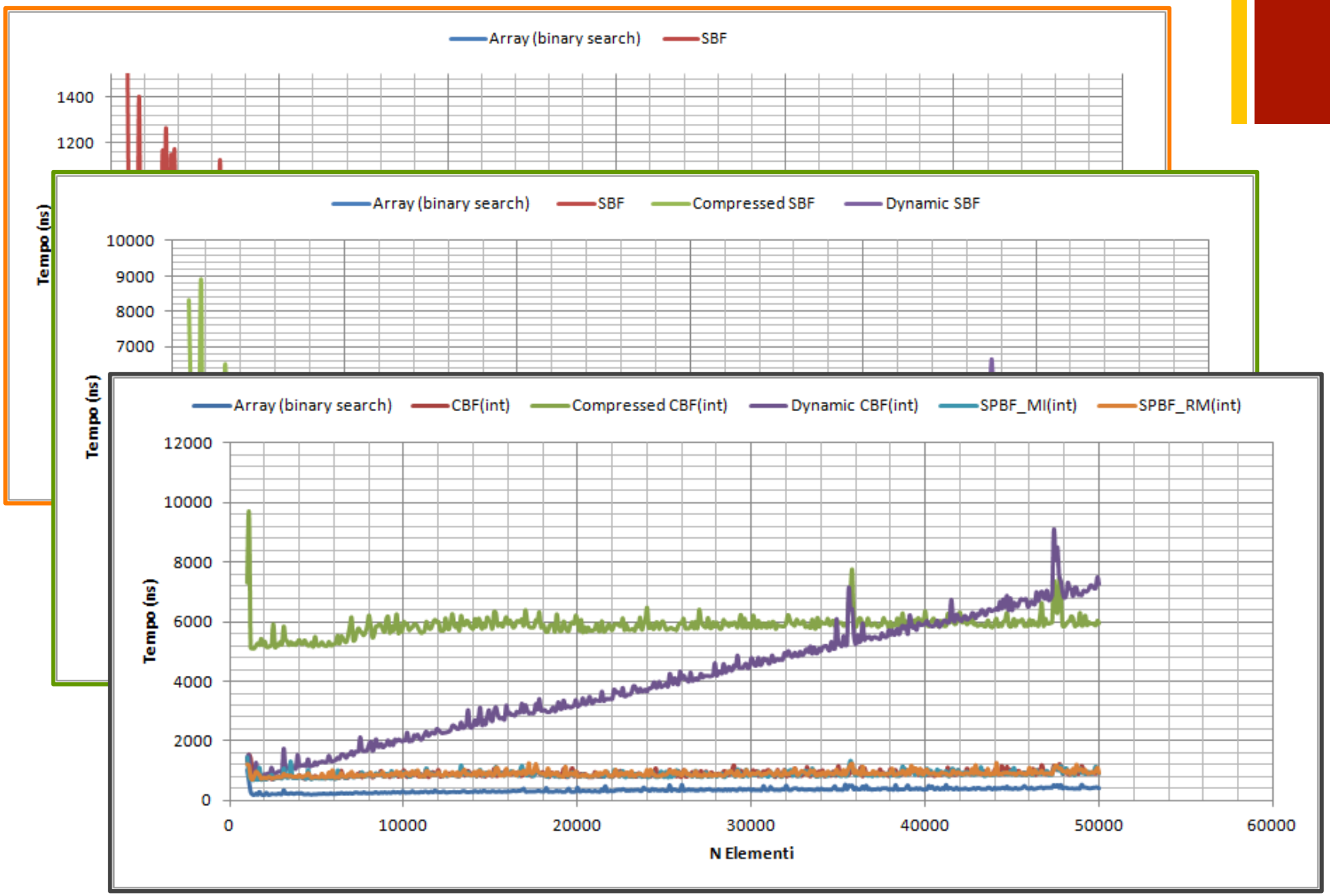
Analisi: Spazio



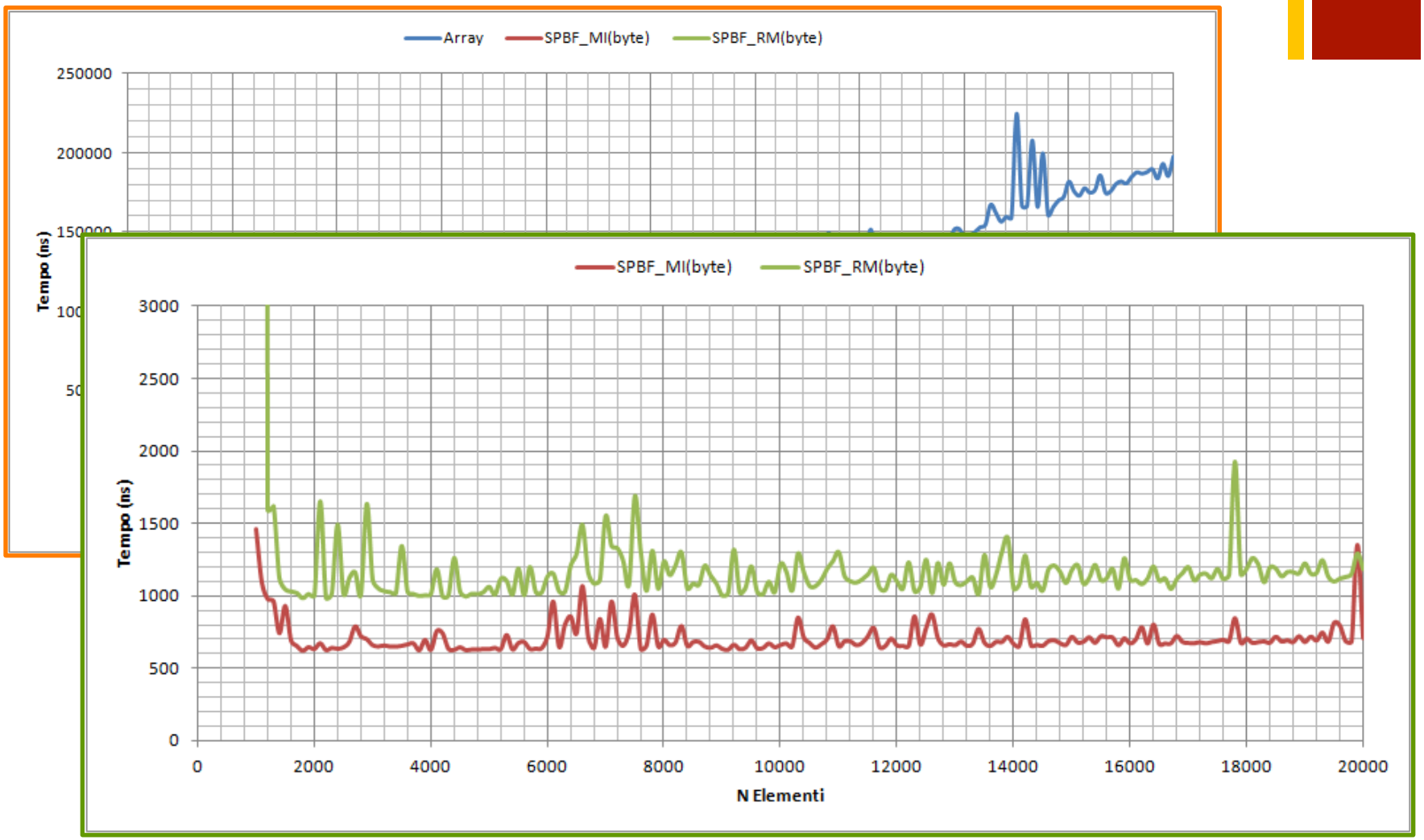
Analisi: add()


SBF
Compressed SBF
Dynamic SBF
CBF(int)
Compressed CBF(int)
Dynamic CBF(int)
SPBF_MI(int)
SPBF_RM(int)
Tempo (ns)
N Elementi
0
10000
20000
30000
40000
50000
60000

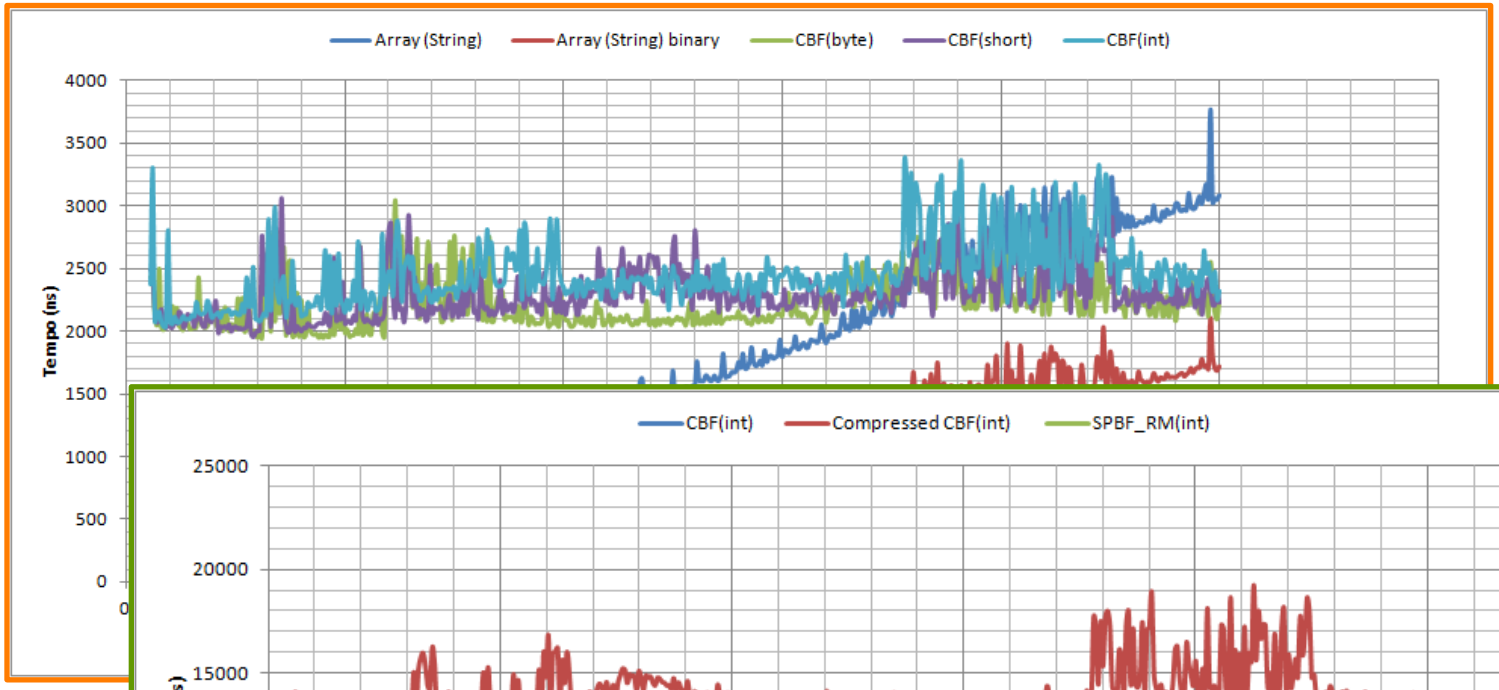
Analisi: lookup()

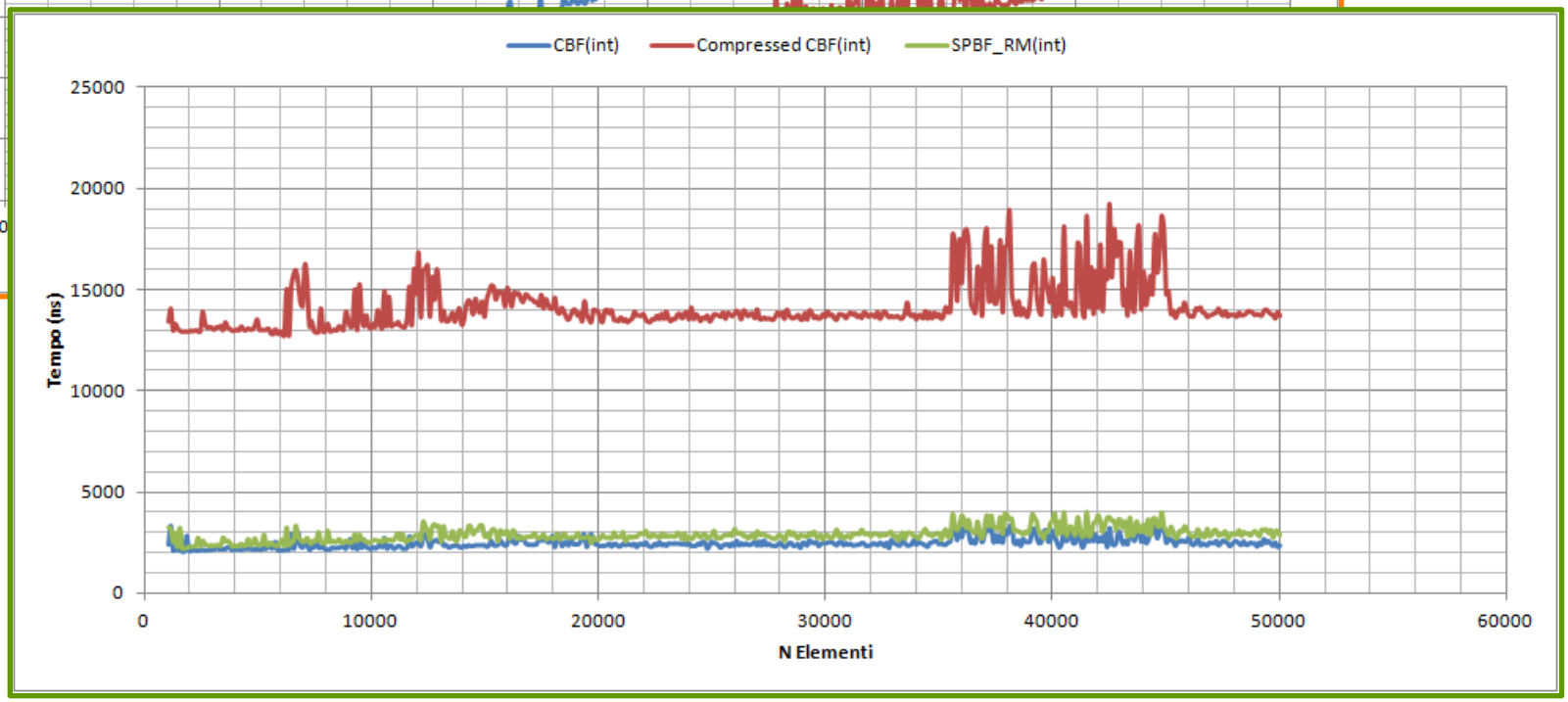


Analisi: frequency()

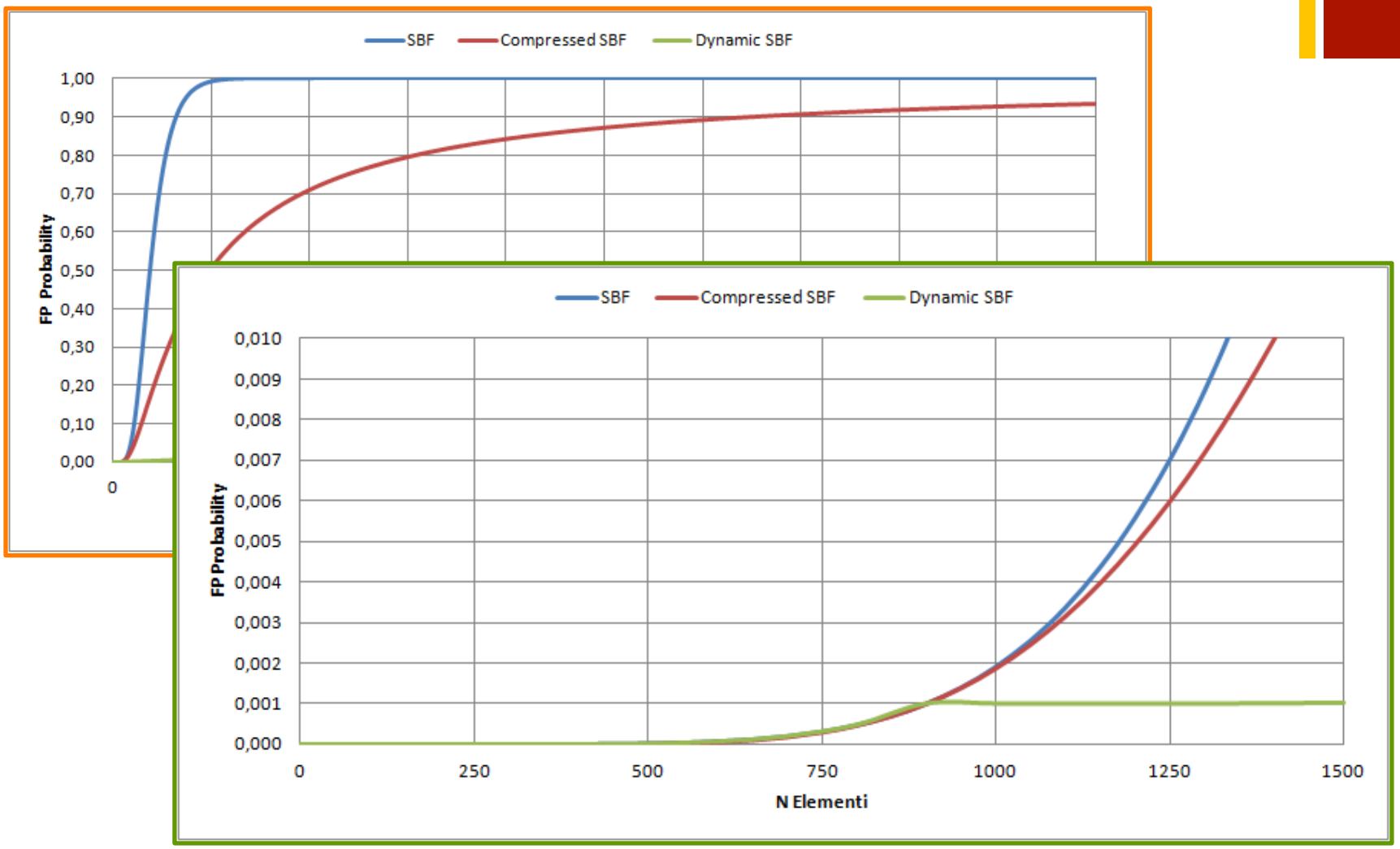


Analisi: delete()





Analisi: FP probability



Minimal Perfect Hash (MPHS)

Soluzione alternativa ai Bloom Filter, che permette di rappresentare un insieme in maniera succinta e di effettuare operazioni di lookup in tempo costante.

- Funzione Hash Minimale Perfetta che mappa l'insieme di n elementi in un range [0, n-1]
- Array di signature di dimensione n, dove ogni elemento occupa 2^{n_bits} dove n_bits dipende dal valore di probabilità desiderata.

$$n\_bits = \log_2 \frac{1}{fp}$$

- Struttura dati statica
- Operazioni consentite:
  - lookup()

Inizializzazione

```
1  public MPHS(List<String> aS,double fp){
2    List<String> list=(List<String>)Arrays.asList(new HashSet<String
         >(aS).toArray(new String[0])); //eliminare i duplicati dalla
         lista
3    this.fp_prob=fp;
4    n=list.size();
5    n_bits=Math.log(1.0/this.fp_prob) / Math.log(2);
6    range=(int)Math.ceil(Math.pow(2,n_bits));
7    signature=new int[n];
8    try {
9      mph= new GOVMinimalPerfectHashFunction.Builder<String>().keys(
           list).transform( TransformationStrategies.utf32() ).signed(
            32).build();
10     for (int i=0;i<n;i++) signature[(int)mph.getLong(list.get(i))]=
           getSignature(list.get(i));
11   } catch (IOException e) {
12     e.printStackTrace();
13   }
14 }
```

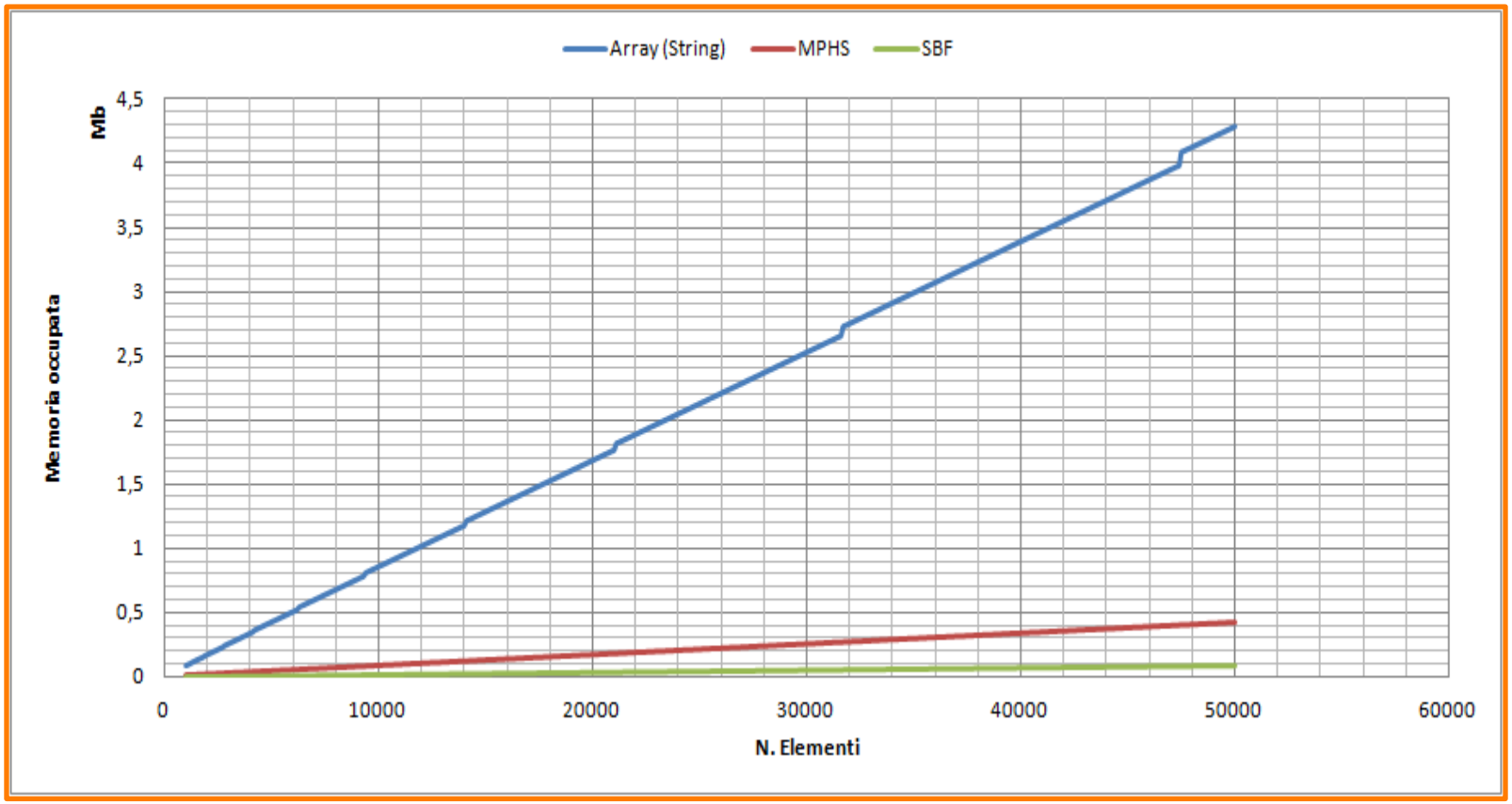
```
1  private int getSignature(Object c){
2    return (int) (c.hashCode() % range);
3  }
```

Lookup(x)

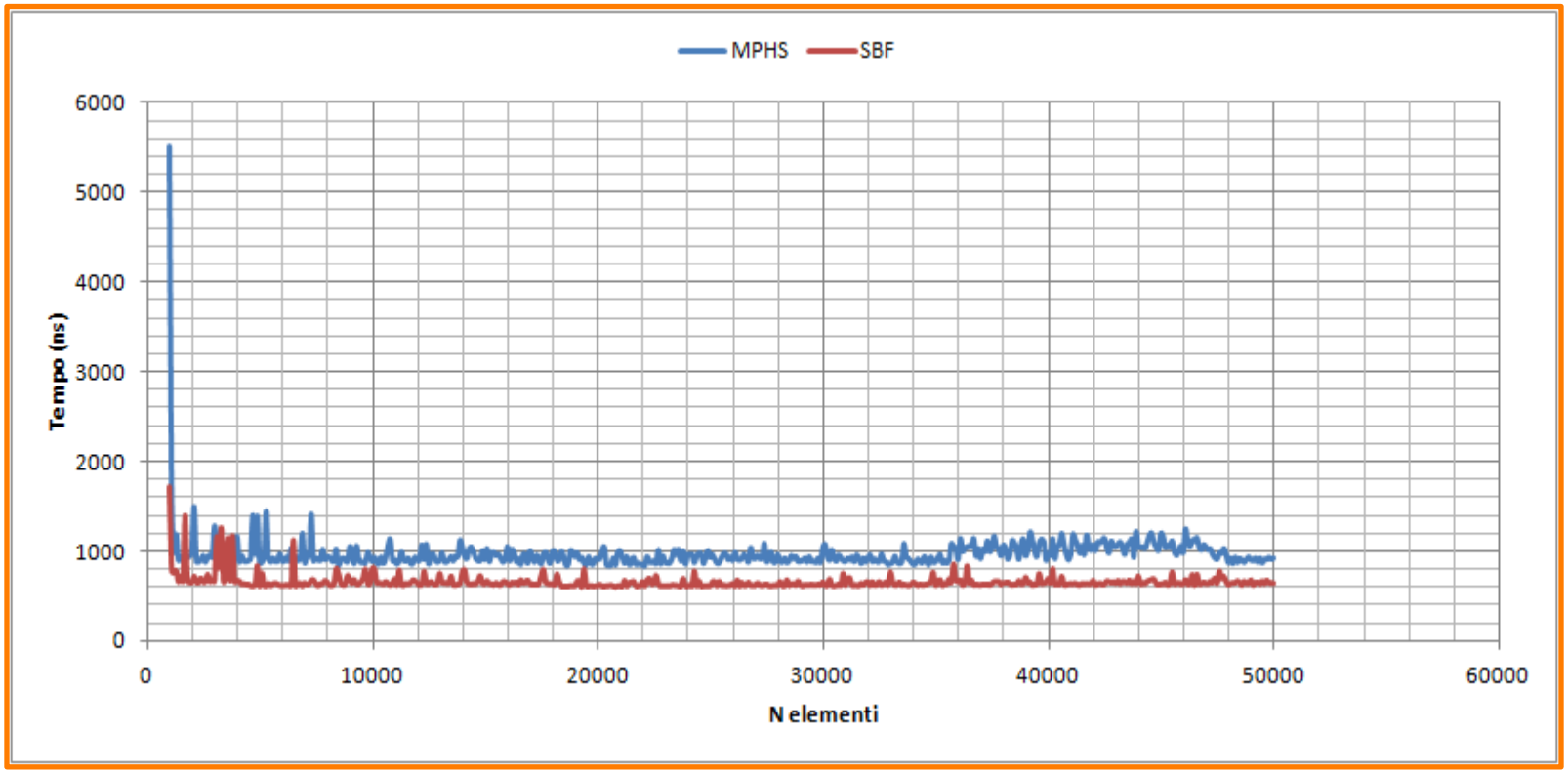
```
public boolean lookup(String s){
    if(mph.getLong(s)!=-1)
        if (getSignature(s)==signature[(int) mph.getLong(s)]) return true;
        else return false;
    else return false;
}
```

Costo: O(k)

MPHS vs SBF: Spazio



MPHS vs SBF: lookup()



Conclusioni

- Bloom Filter sono una buona soluzione
  - per risparmiare spazio
  - eseguire alcune operazioni in tempo costante
  - adattabile alle esigenze dei problemi
- MPHS è una alternativa valida ai BF
  - risparmiare spazio
  - eseguire operazioni di lookup() in tempo costante
- Problema principale: identificare i valori ottimali delle proprietà dei BF per le varie varianti.