Extending STL maps using LBSTs

Leonor Frias

Departament de Llenguatges i Sistemes Informàtics,
Universitat Politècnica de Catalunya

Supported by GRAMMARS project under grant CYCIT TIN2004-07925
and **AEDRI II** project under grant MCYT TIC2002-00190

Standard Template Library (STL)

- Core of C++ standard library
  [International Standard ISO/IEC 14882, 1998]
- STL basic elements:
  - Containers *(list, queue, set, map…)*
    - Store collection of elements
  - Iterators
    - Access and traverse elements in a container
  - Algorithms *(sort, reverse…)*
    - Process container elements

Map

- Efficient access by **key**
- **Ordered** upon key
- Main operations:
  - Insertion by key
  - Search by key
  - Erase by key

  } $\Theta(\log n)$

  - Sequential traversal (using bidirectional iterators)

  } amortized $\Theta(1)$

Map usage example (1)

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

int main(){
    string word;
    map<string,int> M;

    while(cin >> word) ++M[word];

    map<string,int>::iterator it;
    for (it=M.begin(); it!=M.end(); ++it)
       cout<<it->first<<" "<<it->second<<endl;
}
```

Map usage example (2) : get median

```
size_t n= M.size();
string median;

it= M.begin();
for (size_t i=1; i<n/2; ++i) ++it;
median= it->first;
```

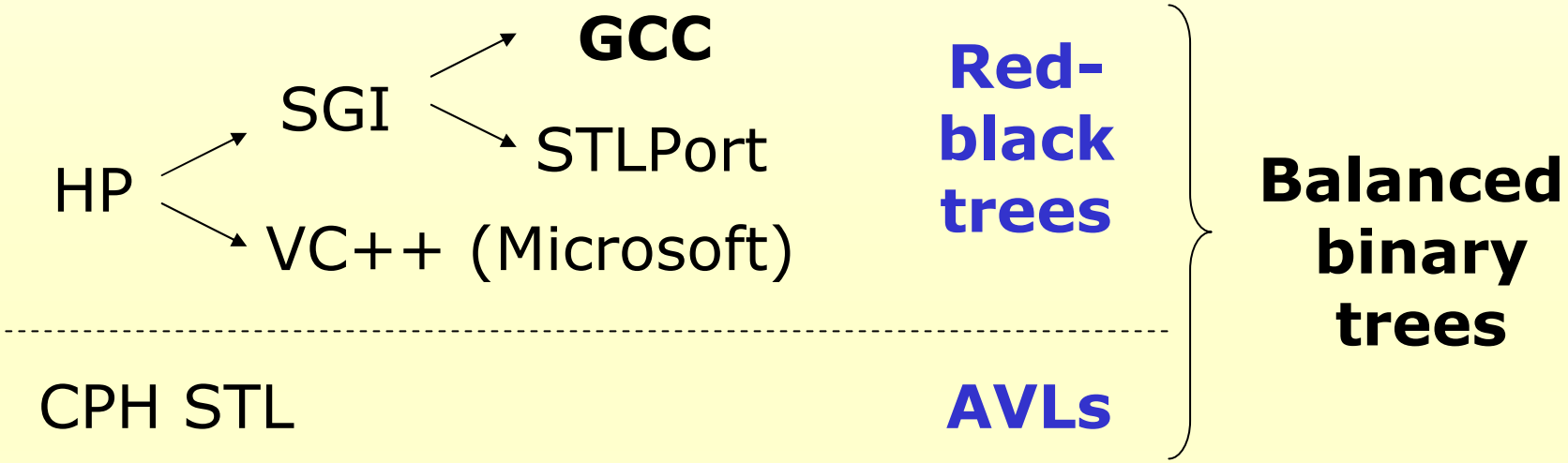
Θ (n)

Efficient access by rank
(with random access iterators)

```
median= M.begin()[n/2].first;
```

Θ (log n)

Implementations for *map*



CPH STL — **B+ trees**

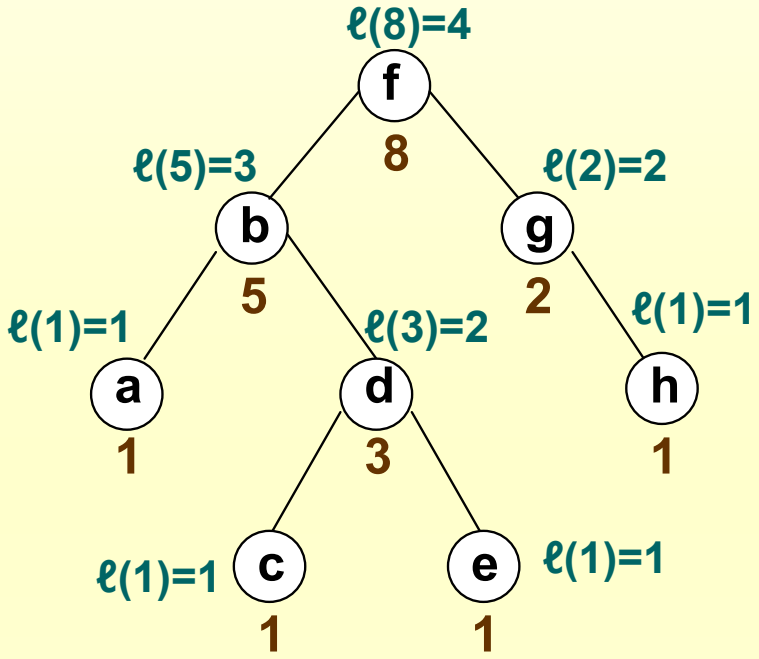
→ Access by rank NOT supported

LBSTs [Roura, ICALP 2001]

- Balancing criteria: subtree sizes
- Efficient rank ops
- Definition:

T is an LBST $\Leftrightarrow$

- T is an empty tree
- T has subtrees L and R, s.t.:
  - they are LBSTs
  - $|\ell(L)- \ell(R)|\leq 1$

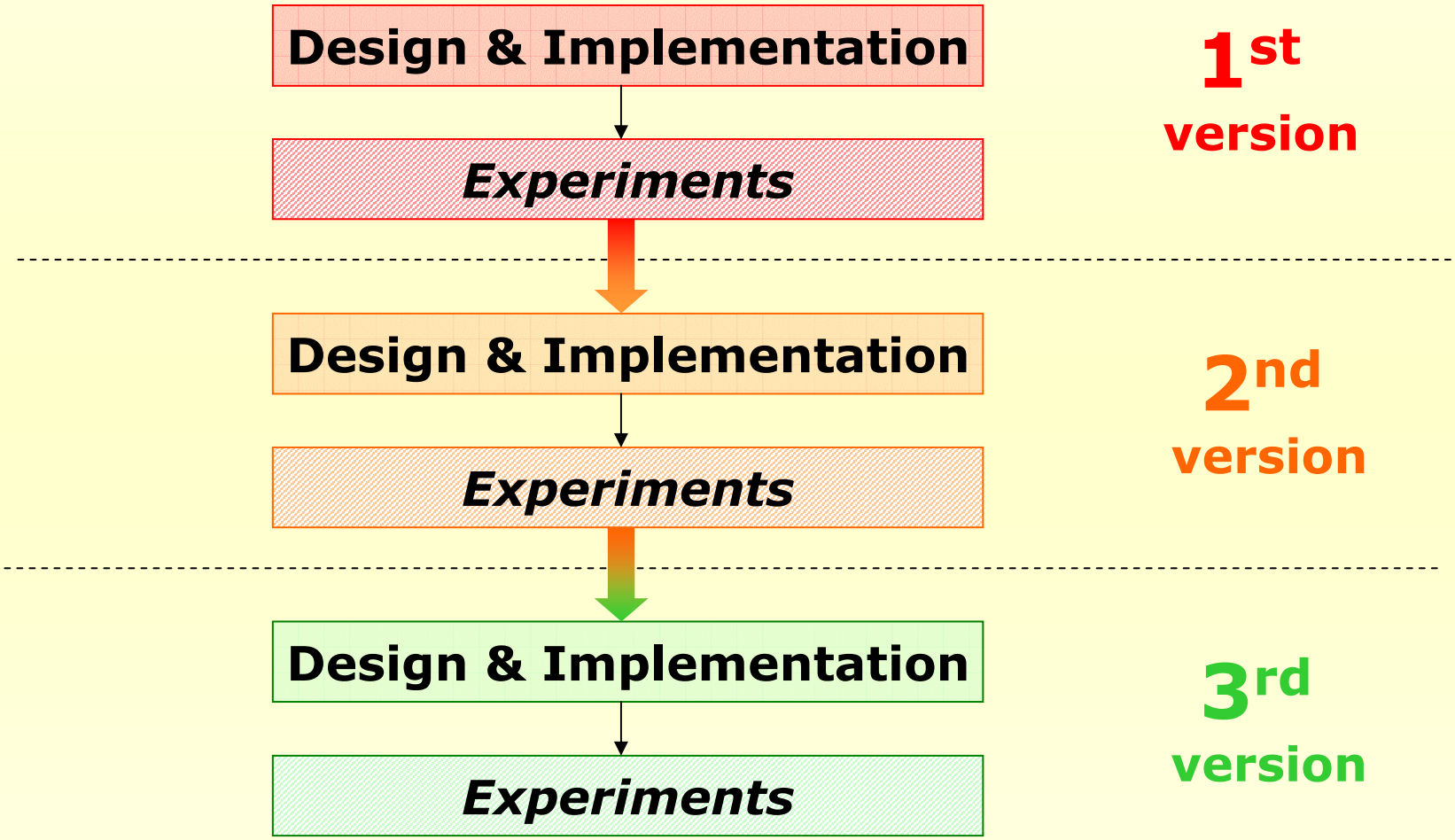


$$\ell(n)= \begin{cases} 0, & \text{if } n=0 \\ 1+\lfloor \log_2 n \rfloor, & \text{if } n\geq 1 \end{cases}$$

LBST implementation

- New functionality: rank operations
  - Offered implementing ***random access iterators*** operations (Cost: Θ(log n))
- Specific for *map* class
- Specialized for operations with sorted ranges → Satisfies cost requirements in contrast to GCC
- Limitation: erase from an iterator requires log time

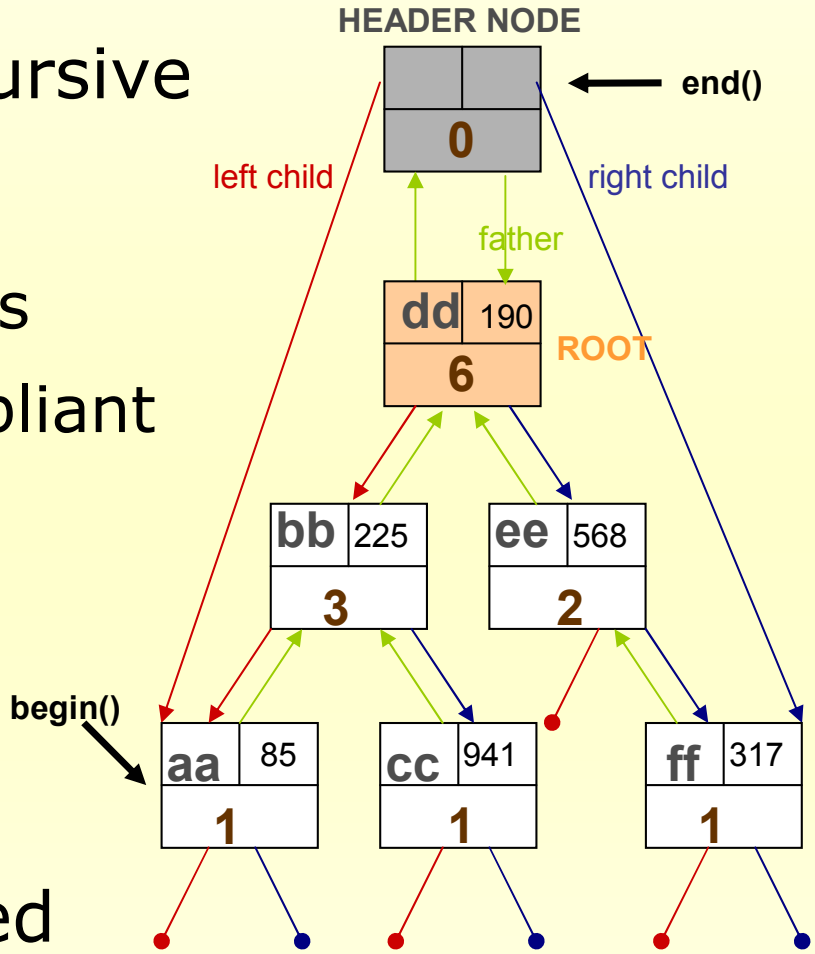
Evolution of the implementation



First version

- Straightforward recursive
- Performance:
  - Slower for most ops
  - BUT standard compliant for sorted ranges
    - Great speedups

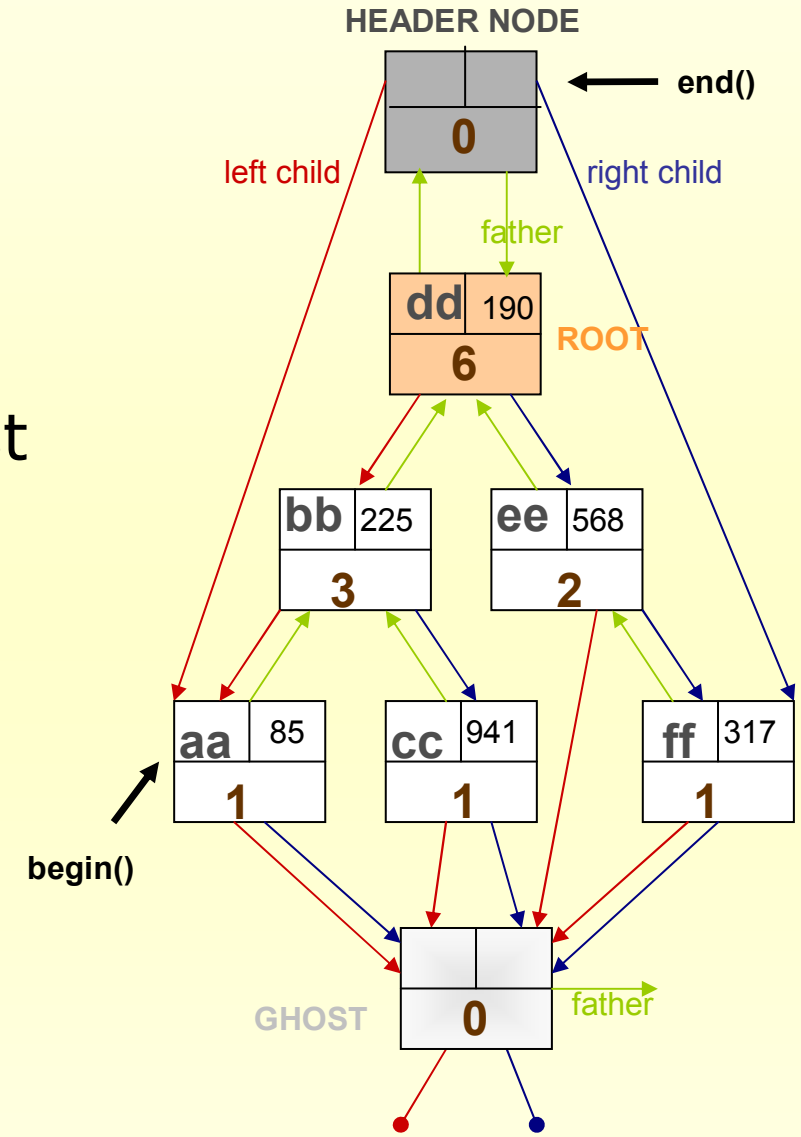
→ Experimental and common knowledge suggested avoiding recursivity



Second version

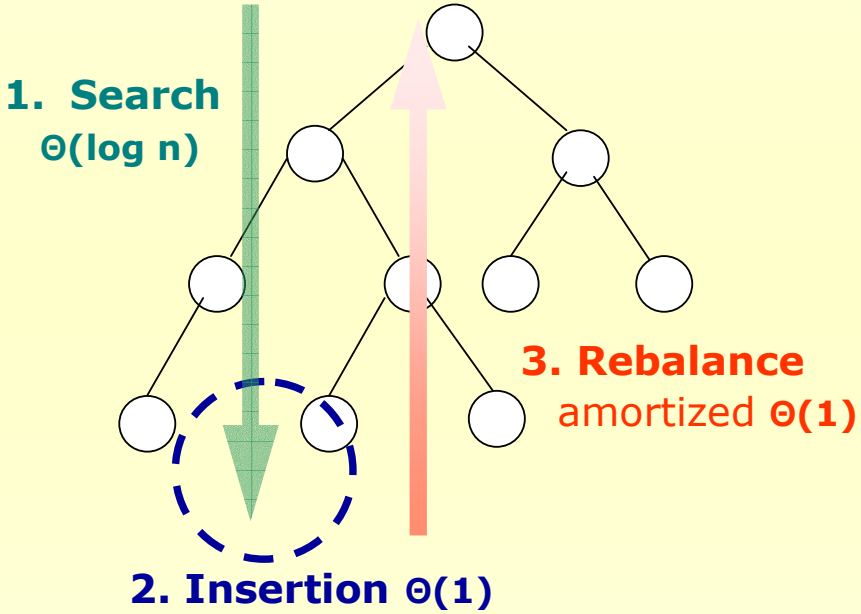
- Iterative
- Performance:
  - Improvement in most operations
  - But simple insertion from random data is still 30% slower

→ Deep analysis of insertion operation

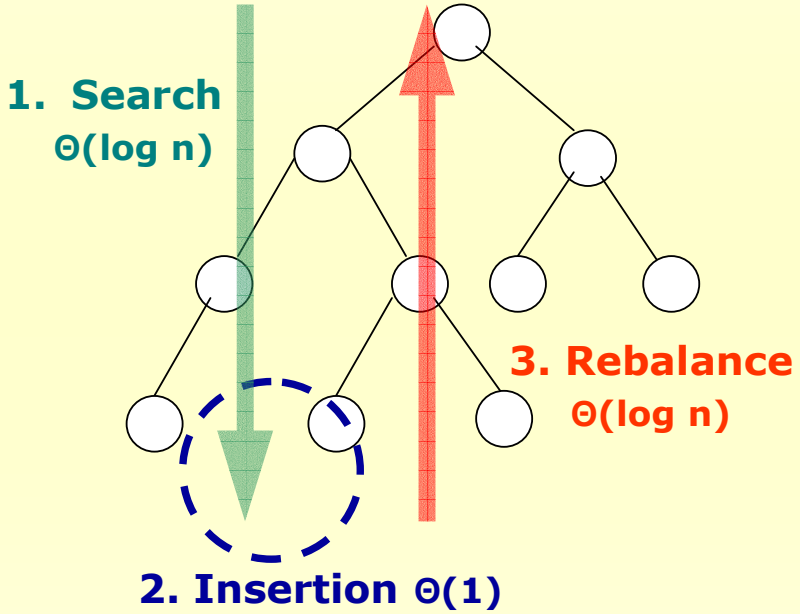


Single insertion/erase analysis

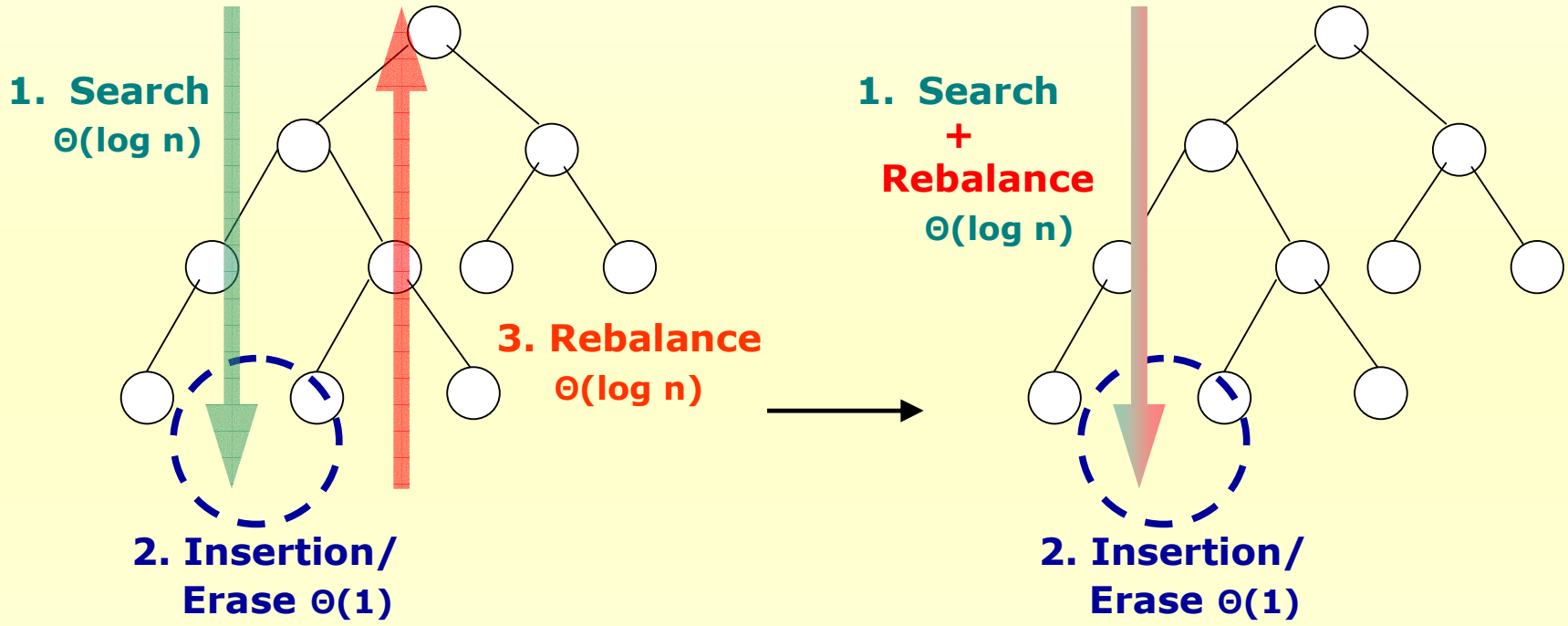
Red-black trees



LBSTs

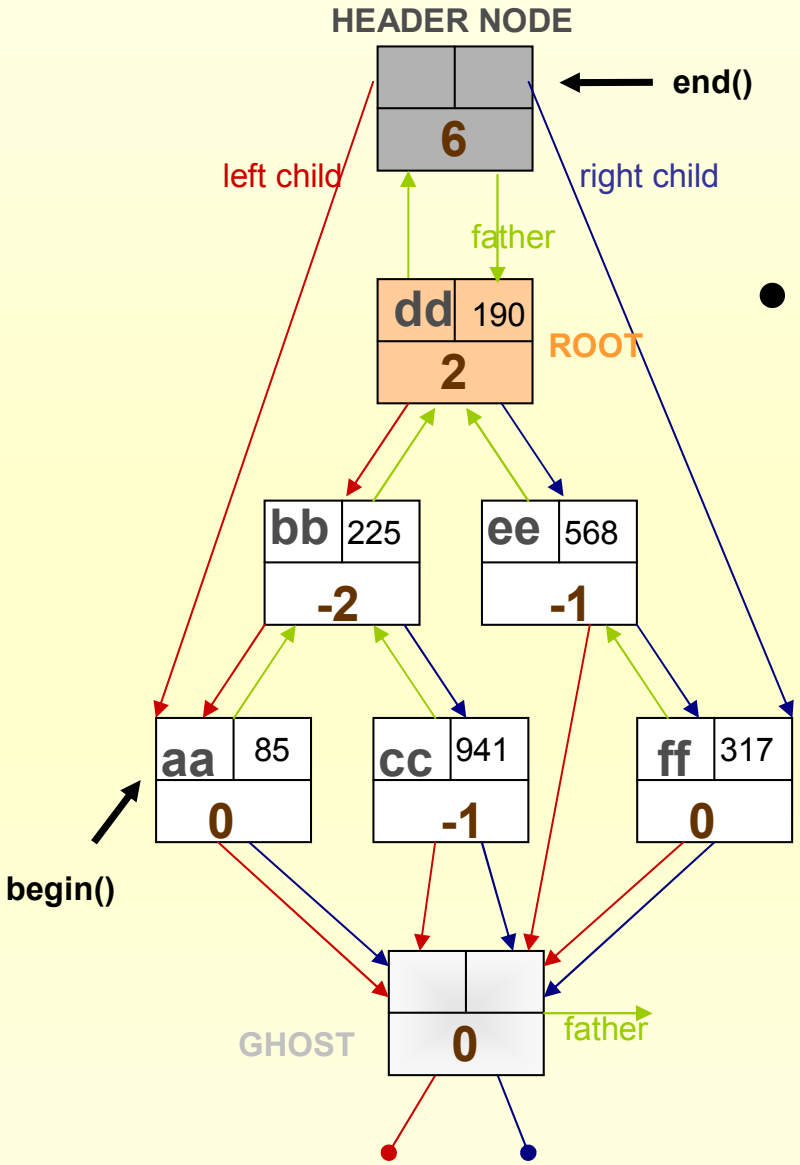


Search & update in a top-down traversal



→ Correctness proved by case analysis

Third version



- Modification of tree nodes:

  Size field (t)=
  t subtree size →
  left /right subtree size

  ↓

  Subtree size must be calculated at every step (only takes **Θ(1)** in avg)

Benefits of the new approach

- New single erase/insertion visits less nodes
- Less size field lookups in search schemas (only at *used* nodes)

less memory accesses

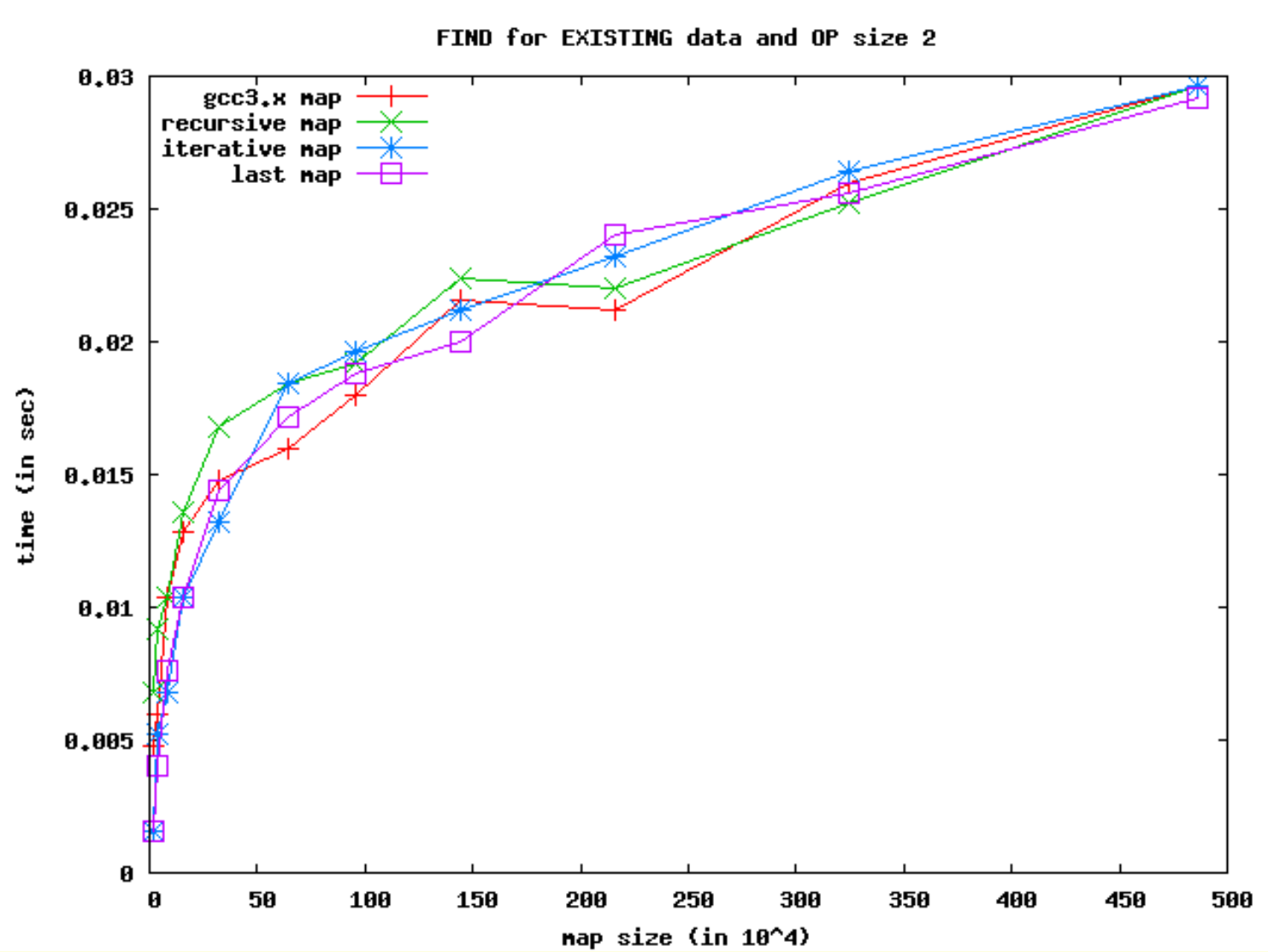
locality improvement

total time reduction

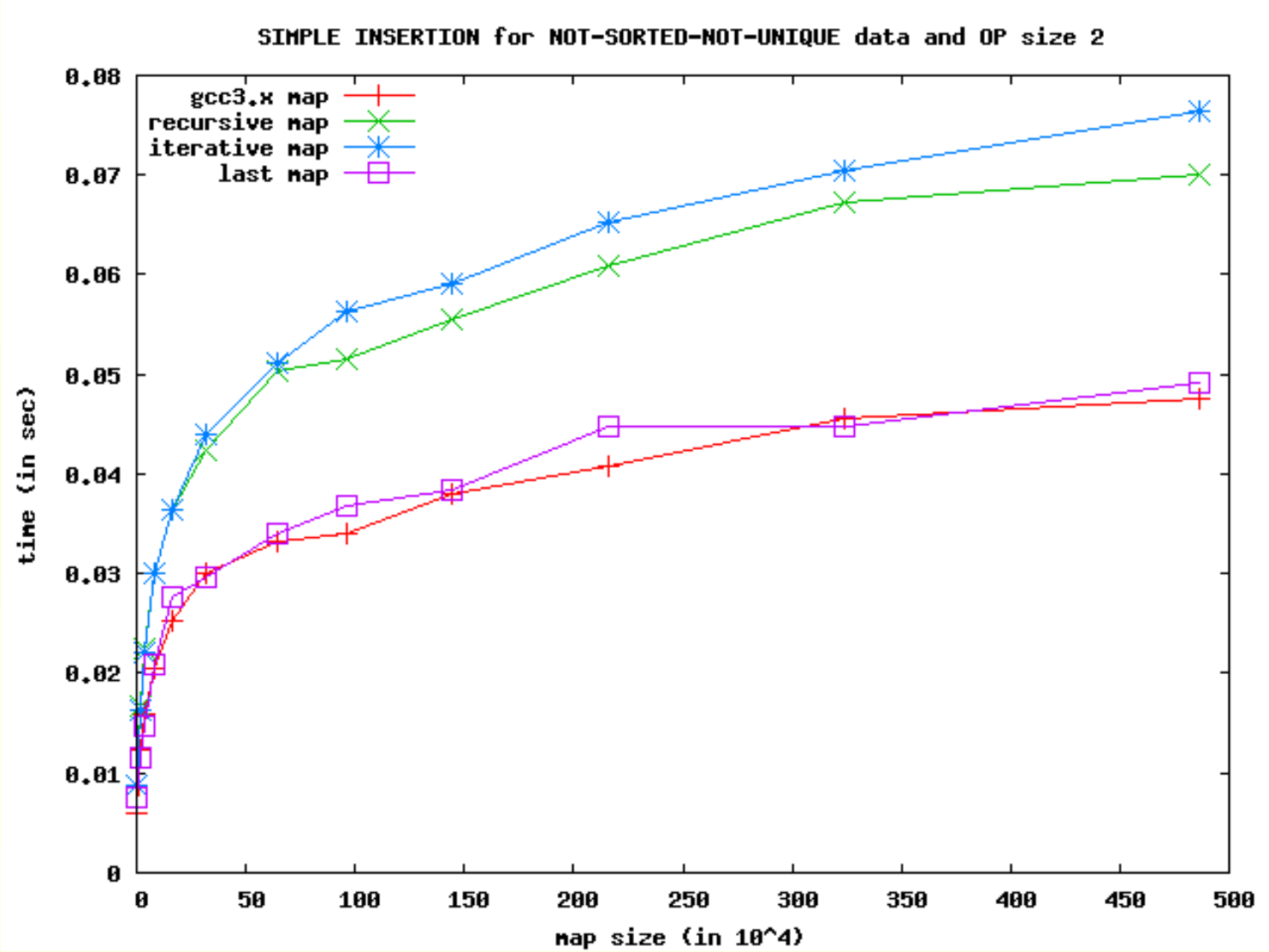
Tests configuration

- Machines:
  - **Intel Pentium4** 3.06 GHz Hyperthreading
    - 512Kb Cache
    - 900 Mb main memory
  - Alpha
  - Ultrasparc
- Parameters
  - Integer & String keys

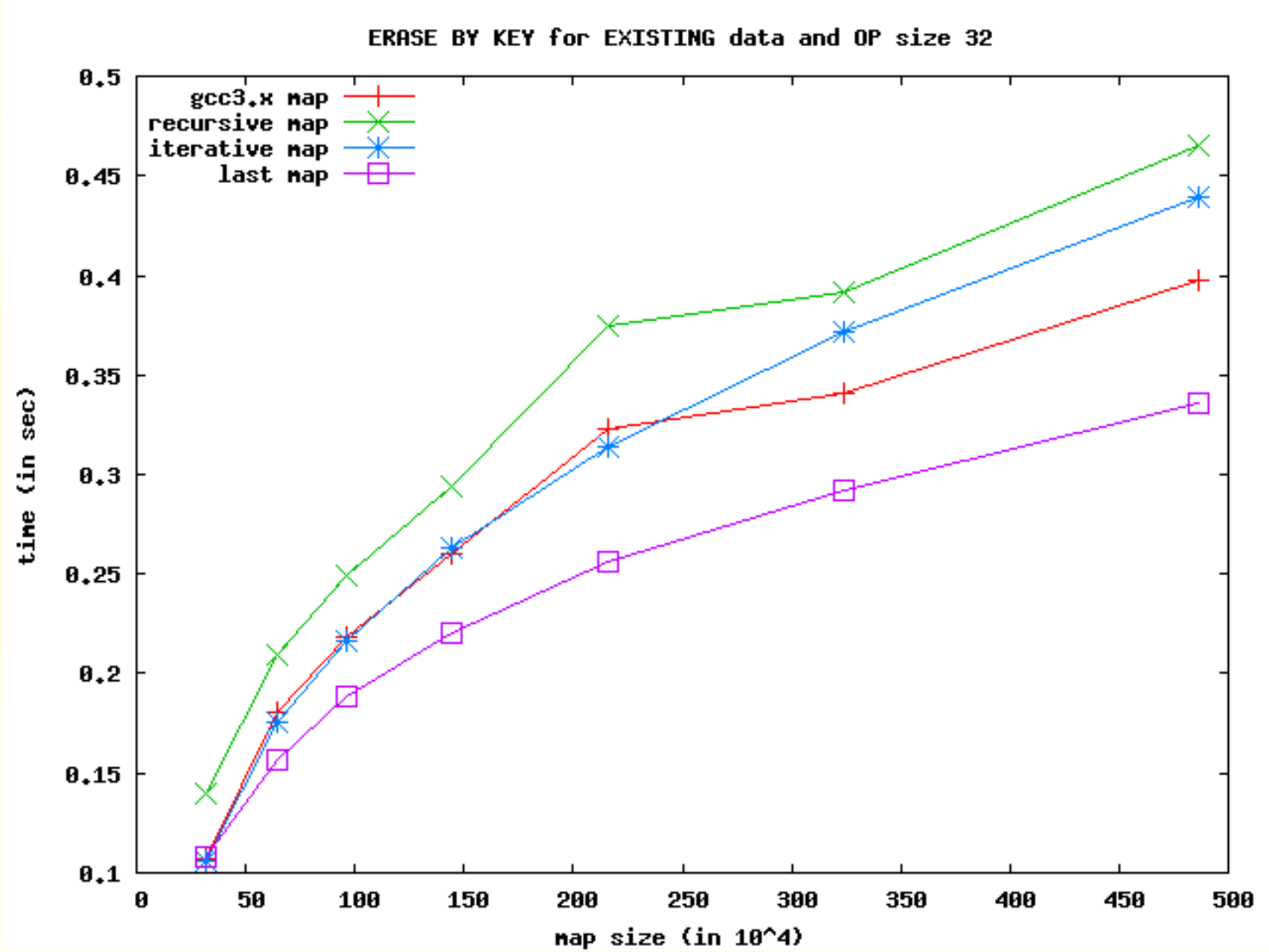
Successful search



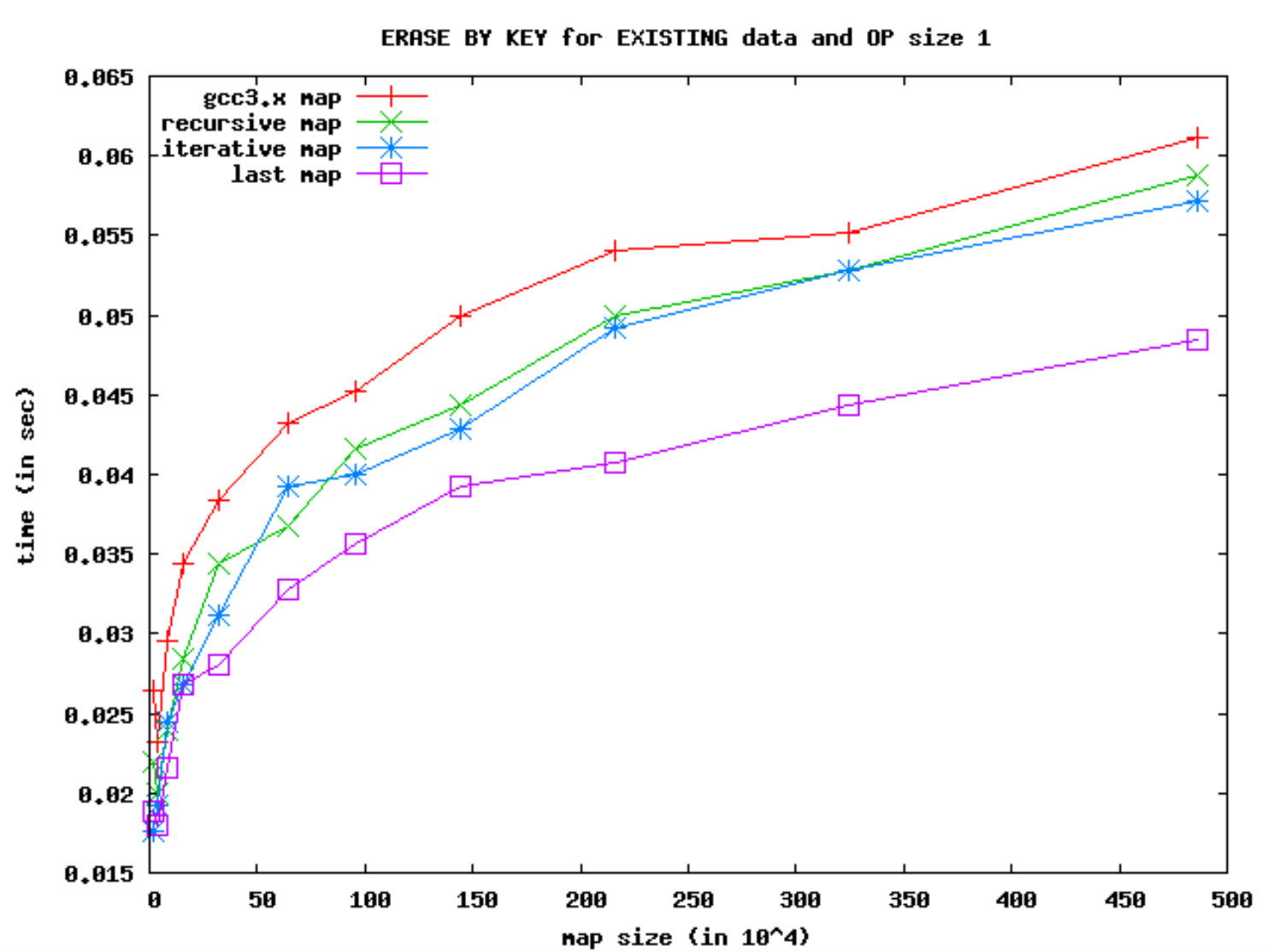
Simple insertion for random data



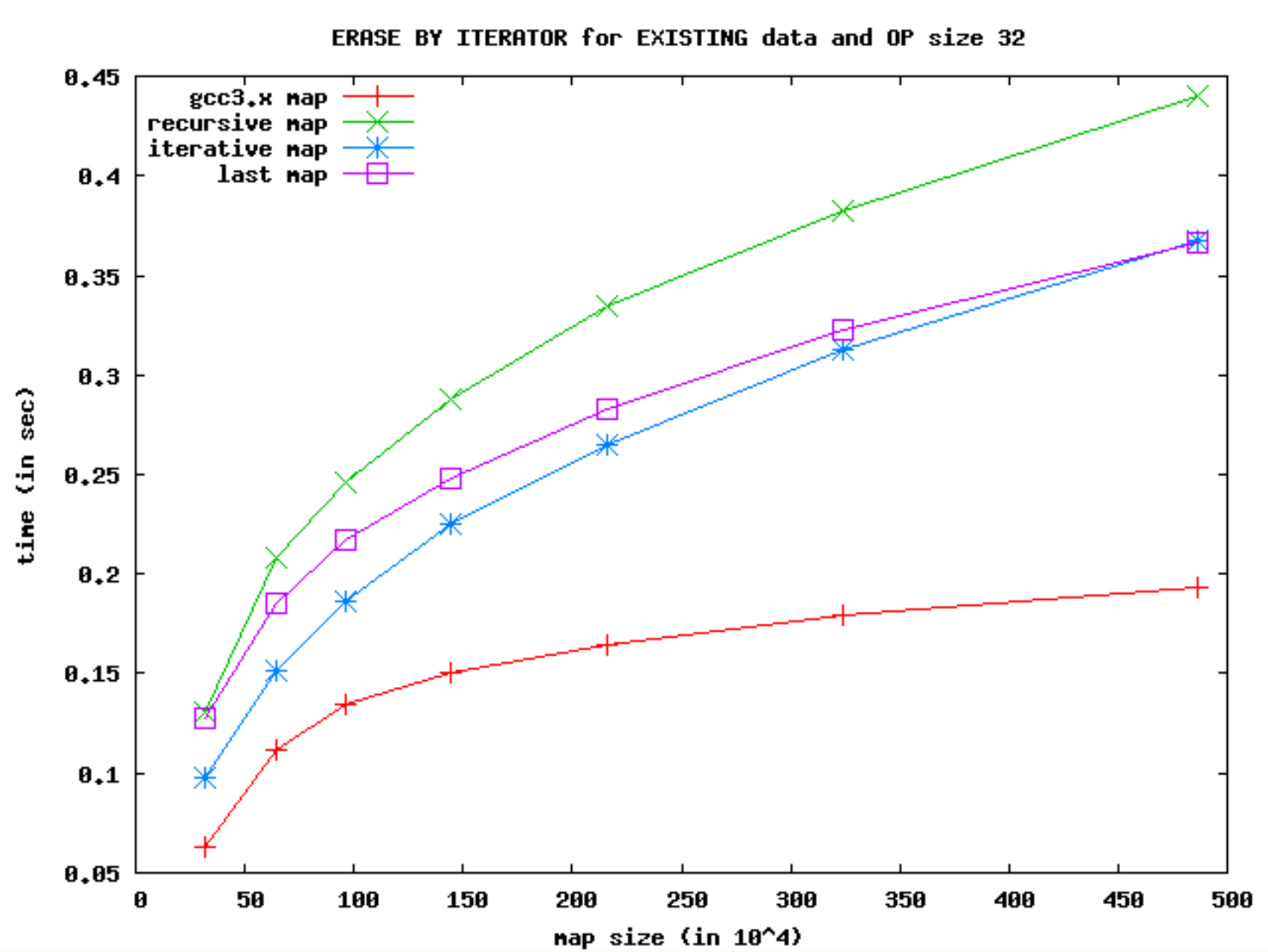
Successful erase by key



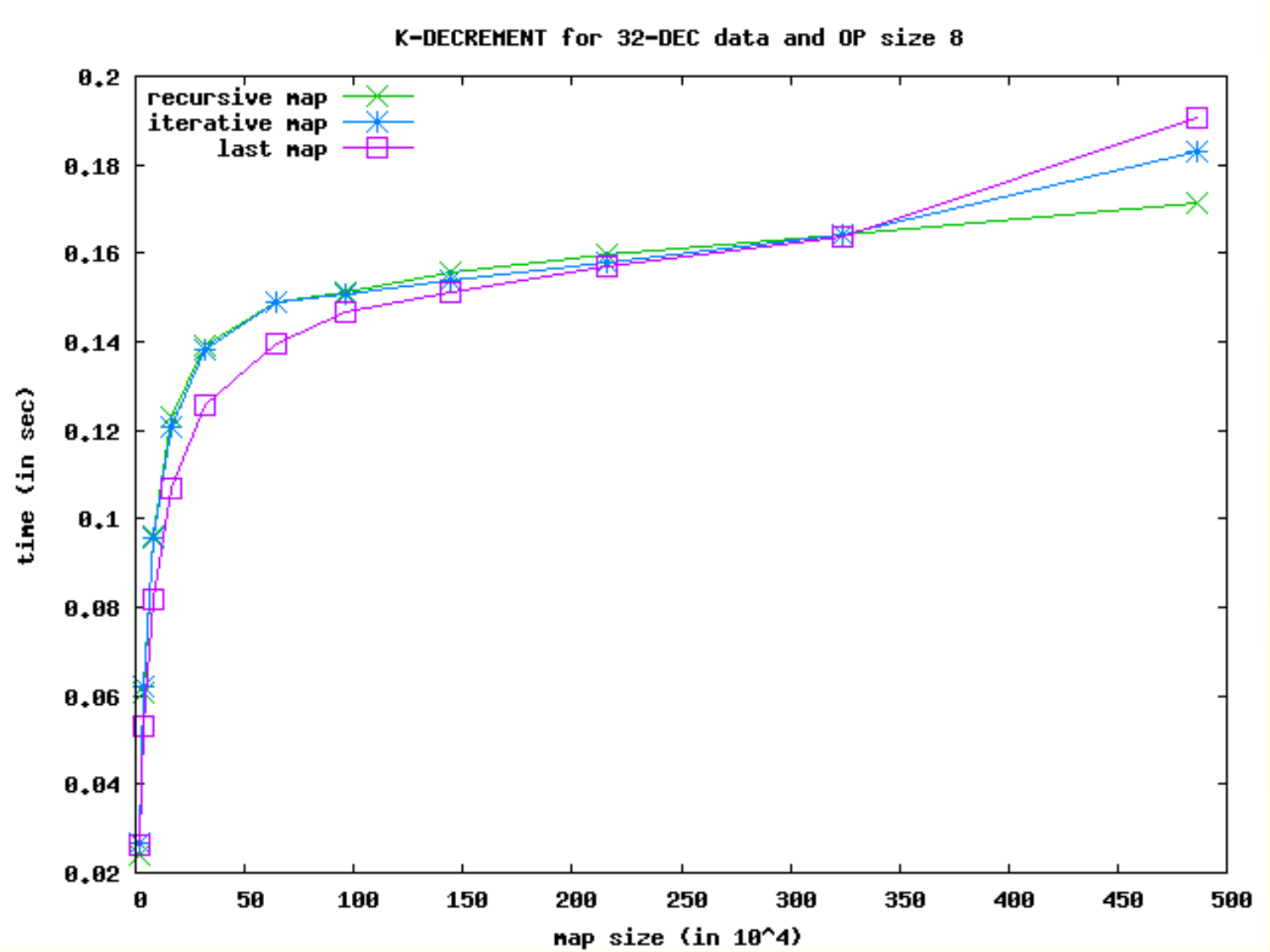
Successful erase by key (*strings*)



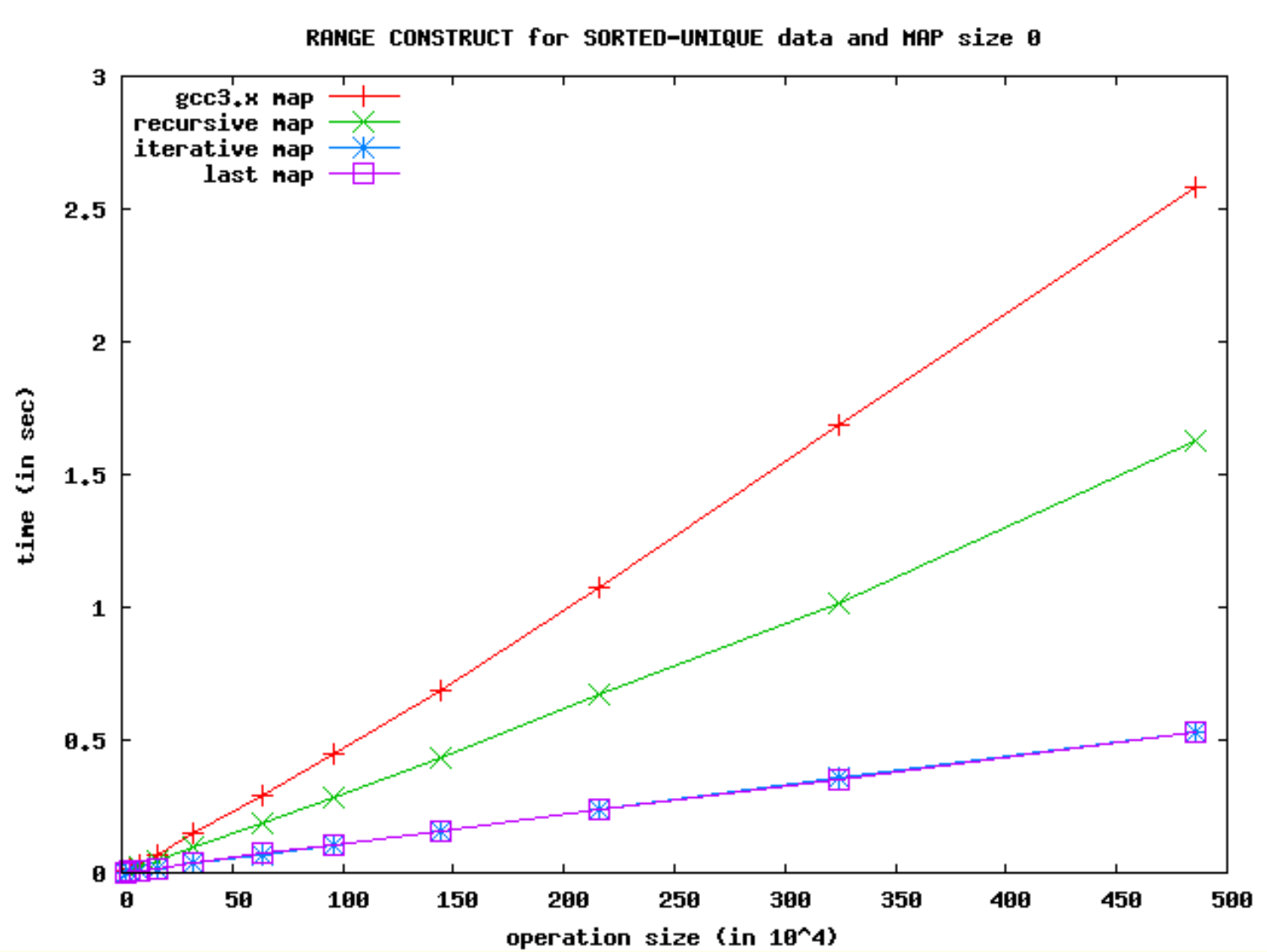
Erase by iterator



32-position decrement



Constructor from a sorted range



Experiments conclusions

- Basic operations { search, insertion, erase } by key
  - Equal or better results than GCC.
- Operations from sorted ranges
  - Compliant with the standard cost requirements
- Erase from an iterator: the only weak point
- k-increment/decrements: new operation

- Results for string keys: equal or better

Conclusions

New implementation of the STL map class

- **Efficient** support to **rank operations**
  - Key point: using **LBSTs**
  - **Standard** mechanism: functionality offered using random access iterators
- **Competitive performance**

Where to find the code & tests

- In my web page
  http://www.lsi.upc.edu/~lfrias/lbst-map/lbst-map.zip

GCC Standard C++ Library

- Description
  http://gcc.gnu.org/onlinedocs/libstdc++/documentation.html

- Code releases
  http://gcc.gnu.org/libstdc++/

Contribute to GCC (Process)

http://gcc.gnu.org/contribute.html

- Legal prerequisites by the Free Software Foundation (FSF)
  - Copyright
- Coding standards
- Testing patches
- Submitting patches

Contribute to GCC (How)

1. Replacing entirely current

2. Make available 2 possible internal implementations

   E.g: Adding an optional template parameter

```
map<key, data, comp, alloc, impl>
```

Insertion with hint

- GCC: one-shot implementation
  http://gcc.gnu.org/onlinedocs/libstdc++/23_containers/howto.html#4
  - Hinting if new element should go **just before the hint**
- LBSTs implementation
  - More general approach
  - Hint always followed

Execution environments

- x86 architecture, Linux OS
  - Intel Pentium 4 CPU 3.06 Ghz hyperthreading
  - Main memory: 900 MB cache: 512 KB
  - Compiler: GCC 3.2
- UltraSparc architecture, sunOS
  - UltraSPARC-II 360 MHz
  - Main memory: 128 MB
  - Compiler: GCC 3.2.3
- Alpha machine, 64-bits
  - Alpha Server 8400
  - Main memory: 2GB
  - Compiler: GCC 3.3