Subject: Re: Map implementation
Posted by mirek on Sun, 14 Apr 2019 17:52:22 GMT
View Forum Message <> Reply to Message

Novo wrote on Sun, 14 April 2019 18:55mirek wrote on Sun, 14 April 2019 02:03If the performance is the issue, then the memory is the issue too. The game starts at L1 cache size, which can correspond to hunderds of records.
How to deal with the memory hierarchy is more or less clear (Cache-Conscious Data Structures, Cache-oblivious algorithms).
What I'm trying to say is that by using a little bit more memory you can significantly improve performance. For example, you can create a bitset of unoccupied slots. That would be overkill for a small hash table.

I made a simple test.

```
  Vector<Index<int> > v;
  Cout() << "sizeof(Index<int>): " << sizeof(Index<int>) << " bytes" << EOL;
  Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
  v.SetCount(v_num);
  Cout() << "Created " << v_num << " empty Index<int>" << EOL;
  Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
  const int isize = 100;
  for (int i = 0; i < isize; ++i) {
   const int jsize = v_num;
   for (int j = 0; j < jsize; ++j)
    v[j].Add(i);
   Cout() << "Added " << i + 1 << " elements" << EOL;
   Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
  }
```

Result:

```
sizeof(Index<int>): 80 bytes
Mem used: 0 Kb
Created 1000000 empty Index<int>
Mem used: 78128 Kb
Added 1 elements
Mem used: 237028 Kb
Added 2 elements
Mem used: 237028 Kb
Added 3 elements
Mem used: 237028 Kb
Added 4 elements
Mem used: 237028 Kb
Added 5 elements
Mem used: 237028 Kb
Added 6 elements
Mem used: 237028 Kb
Added 7 elements
Mem used: 237028 Kb
```

```
Added 8 elements
Mem used: 237028 Kb
Added 9 elements
Mem used: 397796 Kb
Added 10 elements
Mem used: 397796 Kb
...
Added 99 elements
Mem used: 2592740 Kb
Added 100 elements
Mem used: 2592740 Kb
```

IMHO, it is possible to do much better than this ...

Well, keep in mind that lower bound here is 400MB - that is what will cost to store keys itself. If these keys were String, which in reality is much more typical, it would be 1600MB just for values (if they are small).

Current Index overhead is on average ~20 bytes per element, which about matches what we see here. Hard to say you can do much better. E.g. typical 'collisions are linked list' implementation will use about the same number of bytes. Even open addressing will need at least 8 bytes per node if you want to have any meaningful 'payload'.

Anyway, thanks for test. You are right that you can reduce that for very specific scenarios. And next index version will probably do about 20% better.

BTW, test putting 100 * 1000000 elements into single Index will produce the same results.

Mirek

P.S.: Overall I am happy that you both started digging here as I am thinking about refactoring Index, deprecating and detuning some unsused features (ever used FindPrev? :) in favor of those used most frequently.