Integer Algorithms and Data Structures

and why we should care about them

Vladimír Čunát

Department of Theoretical Computer Science and Mathematical Logic

CHARLES UNIVERSITY PRAGUE

faculty of mathematics and physics

Doctoral Seminar 2010/11

Outline

Outline

Motivation

Whan can restriction on integers give us?

mostly comparison based ADS are taught

- except for hashing and radix/bucket sorting
- only pairwise comparisons are assumed
- very general, always usable (where sensible)
- most studied in 70s–80s

but we can often be more restrictive on the keys

- that can give us some benefits

Motivation

Whan can restriction on integers give us?

word size matters

- usual assumption: keys have machine-word size
- significant difference needs different ADS
- longer keys form strings of words
  - pairwise comparisons would need $\Omega(1)$ time (!)
  - we won't discuss ADS for this case
- longer words give us more computational power
  - we can handle multiple keys at once, simulating SIMD

Motivation

Whan can restriction on integers give us?

word size matters

- usual assumption: keys have machine-word size
- significant difference needs different ADS
- longer keys form strings of words
  - pairwise comparisons would need $\Omega(1)$ time (!)
  - we won't discuss ADS for this case
- longer words give us more computational power
  - we can handle multiple keys at once, simulating SIMD

Motivation

Whan can restriction on integers give us?

word size matters

- usual assumption: keys have machine-word size
- significant difference needs different ADS
- longer keys form strings of words
  - pairwise comparisons would need $\Omega(1)$ time (!)
  - we won't discuss ADS for this case
- longer words give us more computational power
  - we can handle multiple keys at once, simulating SIMD

Motivation

Whan can restriction on integers give us?

word size grows quickly

- standard width 64 bits (since $\sim$ 2000)
- SIMD extensions provide operations on longer words
  - the set of operations is restricted but usually sufficient
  - 128-bit words since $\sim$ 2000
  - 256-bit words coming this year (AVX) and expected to grow
- but we often only need 32-bit keys
  $\Rightarrow$ handling 4 or 8 keys at once (today)
- external-memory model: hundreds of keys in block

Motivation

Whan can restriction on integers give us?

word size grows quickly

- standard width 64 bits (since $\sim$ 2000)
- SIMD extensions provide operations on longer words
  - the set of operations is restricted but usually sufficient
  - 128-bit words since $\sim$ 2000
  - 256-bit words coming this year (AVX) and expected to grow
- but we often only need 32-bit keys
  $\Rightarrow$ handling 4 or 8 keys at once (today)
- external-memory model: hundreds of keys in block

Motivation

Whan can restriction on integers give us?

word size grows quickly

- standard width 64 bits (since $\sim$ 2000)
- SIMD extensions provide operations on longer words
  - the set of operations is restricted but usually sufficient
  - 128-bit words since $\sim$ 2000
  - 256-bit words coming this year (AVX) and expected to grow
- but we often only need 32-bit keys
  $\Rightarrow$ handling 4 or 8 keys at once (today)
- external-memory model: hundreds of keys in block

Motivation

Whan can restriction on integers give us?

word size grows quickly

- standard width 64 bits (since $\sim$ 2000)
- SIMD extensions provide operations on longer words
  - the set of operations is restricted but usually sufficient
  - 128-bit words since $\sim$ 2000
  - 256-bit words coming this year (AVX) and expected to grow
- but we often only need 32-bit keys
  $\Rightarrow$ handling 4 or 8 keys at once (today)
- external-memory model: hundreds of keys in block

Outline

Justification

Why can we restrict keys to integers?

discussed ADS will only work with nonnegative integers

- in real computers we have to use them anyway
- we only have to ensure correct ordering

Justification

Why can we restrict keys to integers?

discussed ADS will only work with nonnegative integers

- in real computers we have to use them anyway
- we only have to ensure correct ordering

Justification

Why can we restrict keys to integers?

examples

- negative integers: biased representation works
  (adding half of the nonnegative maximum to all keys)
- IEEE-754 floats: sign – biased exponent – mantissa
  - nonnegative floats compare correctly (!)
  - flipping the sign bit of positive numbers
    and inverting negative ones does the trick (except for NaNs)
- lexicographical ordering of strings:
  by correct alignment or prefixing with length

Justification

Why can we restrict keys to integers?

examples

- negative integers: biased representation works
  (adding half of the nonnegative maximum to all keys)
- IEEE-754 floats: sign – biased exponent – mantissa
  - nonnegative floats compare correctly (!)
  - flipping the sign bit of positive numbers
    and inverting negative ones does the trick (except for NaNs)
- lexicographical ordering of strings:
  by correct alignment or prefixing with length

Justification

Why can we restrict keys to integers?

examples

- negative integers: biased representation works
  (adding half of the nonnegative maximum to all keys)
- IEEE-754 floats: sign – biased exponent – mantissa
  - nonnegative floats compare correctly (!)
  - flipping the sign bit of positive numbers
    and inverting negative ones does the trick (except for NaNs)
- lexicographical ordering of strings:
  by correct alignment or prefixing with length

Justification

Why can we restrict keys to integers?

examples

- negative integers: biased representation works
  (adding half of the nonnegative maximum to all keys)
- IEEE-754 floats: sign – biased exponent – mantissa
  - nonnegative floats compare correctly (!)
  - flipping the sign bit of positive numbers
    and inverting negative ones does the trick (except for NaNs)
- lexicographical ordering of strings:
  by correct alignment or prefixing with length

Outline

The word-RAM model

What can we do with the integers?

we need a model that approximates the power of real HW ⇒

- formalizing our ADS and proving asymptotic complexities
- possibility of proving complexity lower bounds of the problems

word-RAM

- only works with words: w-bit integers (RAM was too strong)
- memory is an addressable array of words
- conditional jumps allow standard control structures
- supports C-like operations on words
  - standard arithmetics $+ - *$ div mod
  - bitwise masks, shifts and boolean operations (not, and, or, xor)
  - but sometimes we're restricted to AC_0 operations (no $*$ ...)

The word-RAM model

What can we do with the integers?

we need a model that approximates the power of real HW ⇒

- formalizing our ADS and proving asymptotic complexities
- possibility of proving complexity lower bounds of the problems

word-RAM

- only works with words: w-bit integers (RAM was too strong)
- memory is an addressable array of words
- conditional jumps allow standard control structures
- supports C-like operations on words
  - standard arithmetics $+ - *$ div mod
  - bitwise masks, shifts and boolean operations (not, and, or, xor)
  - but sometimes we're restricted to AC_0 operations (no $*$ ...)

The word-RAM model

What can we do with the integers?

we need a model that approximates the power of real HW ⇒

- formalizing our ADS and proving asymptotic complexities
- possibility of proving complexity lower bounds of the problems

word-RAM

- only works with words: w-bit integers (RAM was too strong)
- memory is an addressable array of words
- conditional jumps allow standard control structures
- supports C-like operations on words
  - standard arithmetics $+ - *$ div mod
  - bitwise masks, shifts and boolean operations (not, and, or, xor)
  - but sometimes we're restricted to AC_0 operations (no $*$ ...)

The word-RAM model

What can we do with the integers?

we need a model that approximates the power of real HW ⇒

- formalizing our ADS and proving asymptotic complexities
- possibility of proving complexity lower bounds of the problems

word-RAM

- only works with words: w-bit integers (RAM was too strong)
- memory is an addressable array of words
- conditional jumps allow standard control structures
- supports C-like operations on words
  - standard arithmetics $+ - *$ div mod
  - bitwise masks, shifts and boolean operations (not, and, or, xor)
  - but sometimes we're restricted to AC_0 operations (no $*$ ...)

Typical problems

What do we want to do with the integers?

DS for dynamic ordered set maintenance

- dictionary: membership query, insertion, deletion (hash table)
- predecessor problem: min, max, predecessor and successor (predecessor of $x \in U$ is the greatest $y \in S$ such that $y < x$)
- augmented set: rank and select queries (rank is the number of less elements, select is the inverse)
  - augmentation can be done with any monoid ($\rightarrow$ e. g. heaps)

sorting

Typical problems

What do we want to do with the integers?

DS for dynamic ordered set maintenance

- dictionary: membership query, insertion, deletion (hash table)
- predecessor problem: min, max, predecessor and successor (predecessor of $x \in U$ is the greatest $y \in S$ such that $y < x$)
- augmented set: rank and select queries (rank is the number of less elements, select is the inverse)
  - augmentation can be done with any monoid ($\rightarrow$ e. g. heaps)

sorting

Typical problems

What do we want to do with the integers?

DS for dynamic ordered set maintenance

- dictionary: membership query, insertion, deletion (hash table)
- predecessor problem: min, max, predecessor and successor (predecessor of $x \in U$ is the greatest $y \in S$ such that $y < x$)
- augmented set: rank and select queries (rank is the number of less elements, select is the inverse)
  - augmentation can be done with any monoid ($\rightarrow$ e. g. heaps)

sorting

Outline

Utilizing long words

Do we really need SIMD support in HW?

vector computations on multiple keys

- in practice we're given SIMD instructions
- but we can simulate many of them in $\mathcal{O}(1)$ time
  - we reserve one additional bit per key
  - addition: works, overflow can be masked out
  - subtraction: works if the results are nonnegative
  - comparison ($\leq$): via subtraction, result in the reserved bits
  - replication: via multiplication
  - horizontal sum: via multiplication or mod
  - rank: sum of comparison results
  - insertion into sorted vector: rank and bit twiddling
  - and many more...

Utilizing long words

Do we really need SIMD support in HW?

vector computations on multiple keys

- in practice we're given SIMD instructions
- but we can simulate many of them in $\mathcal{O}(1)$ time
  - we reserve one additional bit per key
  - addition: works, overflow can be masked out
  - subtraction: works if the results are nonnegative
  - comparison ($\leq$): via subtraction, result in the reserved bits
  - replication: via multiplication
  - horizontal sum: via multiplication or mod
  - rank: sum of comparison results
  - insertion into sorted vector: rank and bit twiddling
  - and many more...

Utilizing long words

Do we really need SIMD support in HW?

vector computations on multiple keys

- in practice we're given SIMD instructions
- but we can simulate many of them in $\mathcal{O}(1)$ time
  - we reserve one additional bit per key
  - addition: works, overflow can be masked out
  - subtraction: works if the results are nonnegative
  - comparison ($\leq$): via subtraction, result in the reserved bits
  - replication: via multiplication
  - horizontal sum: via multiplication or mod
  - rank: sum of comparison results
  - insertion into sorted vector: rank and bit twiddling
  - and many more...

Utilizing long words

Do we really need SIMD support in HW?

vector computations on multiple keys

- in practice we're given SIMD instructions
- but we can simulate many of them in $\mathcal{O}(1)$ time
  - we reserve one additional bit per key
  - addition: works, overflow can be masked out
  - subtraction: works if the results are nonnegative
  - comparison ($\leq$): via subtraction, result in the reserved bits
  - replication: via multiplication
  - horizontal sum: via multiplication or mod
  - rank: sum of comparison results
  - insertion into sorted vector: rank and bit twiddling
  - and many more...

Utilizing long words

How can we speed up ADS with SIMD?

packed B-tree

- we can maintain a sorted vector of keys
- by adding pointers we can easily implement B-tree
- operations on one vector take $\mathcal{O}(1)$ time
  - ranks and comparisons guide us
  - modification by bit shifting and masking
  - we can even maintain subtree sizes
    $\Rightarrow$ also rank and select queries
- with $b \approx \sqrt{w}$ we have capacity $n \approx (\sqrt{w})^h$
  $\Rightarrow$ height and time: $h \approx \log n / \log \sqrt{w} \approx \log_w n$

Utilizing long words

How can we speed up ADS with SIMD?

packed B-tree

- we can maintain a sorted vector of keys
- by adding pointers we can easily implement B-tree
- operations on one vector take $\mathcal{O}(1)$ time
  - ranks and comparisons guide us
  - modification by bit shifting and masking
  - we can even maintain subtree sizes
    $\Rightarrow$ also rank and select queries
- with $b \approx \sqrt{w}$ we have capacity $n \approx (\sqrt{w})^h$
  $\Rightarrow$ height and time: $h \approx \log n / \log \sqrt{w} \approx \log_w n$

Utilizing long words

How can we speed up ADS with SIMD?

packed B-tree

- we can maintain a sorted vector of keys
- by adding pointers we can easily implement B-tree
- operations on one vector take $\mathcal{O}(1)$ time
  - ranks and comparisons guide us
  - modification by bit shifting and masking
  - we can even maintain subtree sizes
    $\Rightarrow$ also rank and select queries
- with $b \approx \sqrt{w}$ we have capacity $n \approx (\sqrt{w})^h$
  $\Rightarrow$ height and time: $h \approx \log n / \log \sqrt{w} \approx \log_w n$

Range reduction

Can't we make the keys shorter?

decomposition of van Emde Boas

- designed for the predecessor problem
- searching for predecessor of $x \in U$:

  we cut the key in half and first test the high half

  $\begin{cases} \cdot \text{ the matching subset exists and its minimum is less than } x \\ \quad \to \text{ the result is in the subset} \\ \cdot \text{ otherwise we need the maximum of the previous subset} \end{cases}$
- min and max are stored, successor is symmetrical
- insertions and deletions the same way, membership by hashing

Range reduction

Can't we make the keys shorter?

decomposition of van Emde Boas

- designed for the predecessor problem
- searching for predecessor of $x \in U$:

  we cut the key in half and first test the high half

  $$\begin{cases} \cdot \text{ the matching subset exists and its minimum is less than } x \\ \quad \rightarrow \text{ the result is in the subset} \\ \cdot \text{ otherwise we need the maximum of the previous subset} \end{cases}$$
- min and max are stored, successor is symmetrical
- insertions and deletions the same way, membership by hashing

Range reduction

Can't we make the keys shorter?

decomposition of van Emde Boas

- unordered dictionaries inside: hashing in am. exp. $\Theta(1)$ time
- we halve the keys in $\Theta(1)$ time and reasonable space
- we can use recursion:
  - stop on $\Theta(w)$ keys and use balanced trees instead
  - am. exp. time $\mathcal{O}(\log w)$ for any operation (pred. problem)
  - another point of view: halving the paths in a binary trie

Range reduction

Can't we make the keys shorter?

decomposition of van Emde Boas

- unordered dictionaries inside: hashing in am. exp. $\Theta(1)$ time
- we halve the keys in $\Theta(1)$ time and reasonable space
- we can use recursion:
  - stop on $\Theta(w)$ keys and use balanced trees instead
  - am. exp. time $\mathcal{O}(\log w)$ for any operation (pred. problem)
  - another point of view: halving the paths in a binary trie

Combination for predecessor problem

Can we do better?

predecessor problem DS by Andersson

using a layered structure:

1. $\approx \sqrt{\log n}$ levels of range reduction
   give us keys of $\approx w/2^{\sqrt{\log n}}$ bits
2. packed B-trees with branching factor $\approx 2^{\sqrt{\log n}}$
   we need trees of height $\approx \frac{\log n}{\log 2^{\sqrt{\log n}}} = \sqrt{\log n}$
3. balanced trees of height $\approx \sqrt{\log n}$ to reduce space
   so the previous layers only need to store $n/2^{\sqrt{\log n}}$ elements

Combination for predecessor problem

Can we do better?

predecessor problem DS by Andersson

using a layered structure:

1. $\approx \sqrt{\log n}$ levels of range reduction
   give us keys of $\approx w/2^{\sqrt{\log n}}$ bits
2. packed B-trees with branching factor $\approx 2^{\sqrt{\log n}}$
   we need trees of height $\approx \frac{\log n}{\log 2^{\sqrt{\log n}}} = \sqrt{\log n}$
3. balanced trees of height $\approx \sqrt{\log n}$ to reduce space
   so the previous layers only need to store $n/2^{\sqrt{\log n}}$ elements

Combination for predecessor problem

Can we do better?

predecessor problem DS by Andersson

using a layered structure:

1. $\approx \sqrt{\log n}$ levels of range reduction
   give us keys of $\approx w/2^{\sqrt{\log n}}$ bits
2. packed B-trees with branching factor $\approx 2^{\sqrt{\log n}}$
   we need trees of height $\approx \frac{\log n}{\log 2^{\sqrt{\log n}}} = \sqrt{\log n}$
3. balanced trees of height $\approx \sqrt{\log n}$ to reduce space
   so the previous layers only need to store $n/2^{\sqrt{\log n}}$ elements

Combination for predecessor problem

Can we do better?

predecessor problem DS by Andersson

the complexities of operations:

- $\sqrt{\log n}$ queries to hash tables (range reduction)
- traversing packed B-tree of height $\sqrt{\log n}$
- finishing on balanced tree of height $\sqrt{\log n}$

- this immediately gives us sorting in $n\sqrt{\log n}$ expected time

Combination for predecessor problem

Can we do better?

predecessor problem DS by Andersson

the complexities of operations:

- $\sqrt{\log n}$ queries to hash tables (range reduction)
- traversing packed B-tree of height $\sqrt{\log n}$
- finishing on balanced tree of height $\sqrt{\log n}$

- this immediately gives us sorting in $n\sqrt{\log n}$ expected time

Outline

Radix sorting

Can we sometimes sort in linear time?

algorithm

- split the keys into k parts
- use stable counting sort on every part
  in $\mathcal{O}(n + 2^{w/k})$ time and $\mathcal{O}(2^{w/k})$ space

selecting parameters

- choose k such that $n \approx 2^{w/k}$
  so every phase take $\mathcal{O}(n)$ time and space
- we sort in space $\mathcal{O}(n)$ and time $\mathcal{O}(nk) = \mathcal{O}(nw/\log n)$
- this is linear for $n \in 2^{\Omega(w)}$
- compare with Andersson's $\mathcal{O}(n\sqrt{\log n})$ time:
  radix is better for $w \geq \log^{3/2} n$

Radix sorting

Can we sometimes sort in linear time?

algorithm

- split the keys into k parts
- use stable counting sort on every part in $\mathcal{O}(n + 2^{w/k})$ time and $\mathcal{O}(2^{w/k})$ space

selecting parameters

- choose k such that $n \approx 2^{w/k}$ so every phase take $\mathcal{O}(n)$ time and space
- we sort in space $\mathcal{O}(n)$ and time $\mathcal{O}(nk) = \mathcal{O}(nw/\log n)$
- this is linear for $n \in 2^{\Omega(w)}$
- compare with Andersson's $\mathcal{O}(n\sqrt{\log n})$ time: radix is better for $w \geq \log^{3/2} n$

Radix sorting

Can we sometimes sort in linear time?

algorithm

- split the keys into k parts
- use stable counting sort on every part
in $\mathcal{O}(n + 2^{w/k})$ time and $\mathcal{O}(2^{w/k})$ space

selecting parameters

- choose k such that $n \approx 2^{w/k}$
so every phase take $\mathcal{O}(n)$ time and space
- we sort in space $\mathcal{O}(n)$ and time $\mathcal{O}(nk) = \mathcal{O}(nw/\log n)$
- this is linear for $n \in 2^{\Omega(w)}$
- compare with Andersson's $\mathcal{O}(n\sqrt{\log n})$ time:
radix is better for $w \geq \log^{3/2} n$

Conclusion

What have we found out about integer ADS?

integer ADS

- comparison model is very simple and general
  but in reality we (can) usually use integer keys
- we can gain significant performance improvement
- comparison model bounds can be broken:
  $\Omega(\log n)$ predecessor, $\Omega(n \log n)$ sorting
- many of the techniques are very useful even in practice,
  for example hashing and radix sorting

Conclusion

What have we found out about integer ADS?

integer ADS

- comparison model is very simple and general
  but in reality we (can) usually use integer keys
- we can gain significant performance improvement
- comparison model bounds can be broken:
  $\Omega(\log n)$ predecessor, $\Omega(n \log n)$ sorting
- many of the techniques are very useful even in practice,
  for example hashing and radix sorting

Further reading

Where to find more about it?

- there are many publications, even asympotical improvements are known, but very complicated
- good introduction, overview and references:

Eric Demaine
Advanced Data Structures
MIT Lecture Notes, 2003, 2005, 2010
http://erikdemaine.org