Putting your data structure on a diet

Jyrki Katajainen (University of Copenhagen)

Joint work Hervé Brönnimann (Polytechnic University) and Pat Morin (Carleton University)

These slides are available at http://www.cphstl.dk
Memory overhead

- The amount of storage used by a data structure beyond what is actually required to store the elements manipulated (measured in words and/or in elements)

- We assume that pointers and integers occupy one word, and elements one or more words still being constant-sized objects

Example: Circular list of $n$ apples; memory overhead $2n + O(1)$ words

$n$: # of elements currently stored
Research question

Q: How much can the memory overhead of a data structure be reduced without destroying its desirable properties?

A: Many data structures can be put on a diet so that, if the original memory overhead is $O(n)$, the memory overhead can be reduced to $O(n/\lg n)$, $\varepsilon n$, or $(1 + \varepsilon)n$ for any $\varepsilon > 0$ and sufficiently large $n > n(\varepsilon)$. The operations on the data structures are not slower, except by a small $O(1)$ factor or/and an additive term of $(1/\varepsilon)$.

True, for example, for

- lists (left as an exercise in the paper)
- ordered dictionaries (considered today)
- priority queues (presented in the paper)
Motivation

According to an earlier study [Brönnimann & Katajainen 2006], a red-black tree that has small memory overhead is faster than the implementation available at the C++ standard library for most operations. For further details, see [CPH STL Report 2006-1]

**Performance ratio:** Our programs were up to 1.2 times faster

Our ultimate goal is to develop library components that guarantee optimal time and space bounds
Focus in this presentation

- Generality of the compaction technique
- Concrete examples

For technical details, see the forthcoming CPH STL report
Memory fragmentation

Allocation of memory segments of varying size can be problematic!

**Internal fragmentation:** Memory space allocated but not used

**External fragmentation:** Memory space that cannot be used because of disadvantageous allocation of memory segments

- □ allocated
- □ wasted due to internal fragmentation
- □ wasted due to external fragmentation
Minimum storage usage

Implicit data structures assume that there is an infinite array available to be used for storing elements; in practice, a resizable array should be used instead.

**Lower bound:** A resizable array requires at least $\Omega(\sqrt{n})$ extra space for pointers and/or elements [Brodnik et al. 1999]

**Upper bound:** Realizations exist that require $O(\sqrt{n})$ extra space. Under a realistic model of dynamic memory allocation, the waste of memory due to internal fragmentation is $O(\sqrt{n})$ [Brodnik et al. 1999], even though external fragmentation can be large.
Earlier approaches

**Ad-hoc designs:** Improve the space efficiency of some specific data structures

**Implicit data structures:** Reduce the memory overhead to $O(1)$ words or $O(\lg n)$ bits

Often the developed data structures, like the searchable heap of Franceschini and Grossi [2003],

- are complicated,
- support a restricted set of operations, and
- do not provide certain desirable properties.
General data-structural transformation

\[ \mathcal{D} \] \hspace{1cm} n \text{ elements} \hspace{1cm} \mathcal{D}'

**Memory overhead:** \( O(n/\lg n), \) \( \varepsilon n, \) or \((1 + \varepsilon)n\) words for any \( \varepsilon > 0 \) and \( n > n(\varepsilon) \)

**Basic idea:** Instead of operating on elements themselves, operate on groups—chunks—of \( O(1/\varepsilon) \) elements
Doubly-linked lists

$D \quad \rightarrow \quad \downarrow \quad \rightarrow$  

1 bit indicates the type of a node (last or not)

$D'$  

$0 \quad 0 \quad 0 \quad 1$

Memory overhead: $n + 3n/b + O(1)$ words, provided that bits can be packed in pointers
Bidirectional iterators: Iterator ++ is an additive term of $O(b)$ slower
Key-based/location-based access

A data structure is called **elementary** if it only supports **key-based access**.

An important requirement often imposed by modern libraries is to provide **location-based access** to elements, as well as to provide **iterators** to step through a set of elements.
Locators and iterators

A **locator** is a mechanism for maintaining the association between an element and its location in a data structure.

An **iterator** is a generalization of a locator that captures the concepts *location* and *iteration* in a container of elements.

Valid expressions:

\[
X \ p; \ X \ p = q; \\
X& \ r = p; \ *p = x; \ x = *p; \ p == q; \ p != q;
\]

**Bidirectional iterators:** Locator expressions plus ++p and --p
Red-black trees

template <typename E>
struct node {
    node* child[2];
    node* parent;
    bool colour;
    E element;
};

Memory overhead: $4n + O(1)$ words or more, because of word alignment

Immediate improvement: Pack the colour bits in pointers $\Rightarrow 3n + O(1)$ words [CPH STL Report 2006-1]
Child-sibling representation

- **x** left child; sibling exists
- **x** has left child
- Store left child & right sibling
- Access parent via sibling
- Access right child via left child

- **x** left child; sibling exists
- **x** has no left child
- Store right child & right sibling
- Access parent via sibling

- **x** left child; no sibling exists
- **x** has left child
- Store left child & parent
- Access right child via left child
Child-sibling representation (cont.)

- $x$ left child; no sibling exists
- $x$ has no left child
- Store right child & parent

- $x$ right child
- $x$ has left child
- Store left child & parent
- Access right child via left child

- $x$ right child
- $x$ has no left child
- Store right child & parent
Child-sibling representation (cont.)

- 3 bits to indicate the type of a node
- 1 bit to indicate the colour of a node

**Memory overhead:** $2n + O(1)$ words, provided that the bits can be packed in pointers
Elementary dictionaries

Store the whole dictionary in an infinite array

$D$:  
- $S(n)$ and $U(n)$ time per search and update  
- Memory overhead of $O(n)$ words  
- All regularity requirements fulfilled

$D'$:  
- $S(n/\lg n) + O(\lg \lg n)$ and $O(S(n/\lg n)+U(n/\lg n)+\lg n)$ per search and update  
- Exactly $n$ locations for elements and at most $O(n/\lg n)$ locations for pointers and integers; furthermore, the whole dictionary can occupy a contiguous segment of memory

**Nice theory:** Freely movable data structures (e.g. circular array); $D'$ works equally well for sets and multisets
Dictionaries with few iterators

\( D: \)
- \( S(n) \) and \( U(n) \) time per key-based/location-based search and update
- Memory overhead of \( O(n) \) words
- Iterator operations in \( O(1) \) time

\( D': \)
- \( O(S(n/b) + \lg b) \) and \( O(S(n/b) + U(n/b) + b) \) time per key-based/location-based search and update
- Memory overhead of \( O(k + n/b) \) where \( k \) is the number of elements currently referenced by iterators
- Iterator operations in \( O(1) \) time
Proof by picture

If elements are moved, update handles inside the iterators

\[ O(n/b) \text{ words} \]

\[ O(n/b) \text{ headers} \]

\[ b \ldots 4b \text{ elements per array; elements in sorted order} \]
Dictionaries with many iterators

$D$:
- $S(n)$ and $U(n)$ time per key-based/location-based search and update
- Memory overhead of $O(n)$ words
- Iterator operations in $O(1)$ time

$D'$:
- $O(S(n/b) + \lg b)$ and $O(S(n/b) + U(n/b) + b)$ time per key-based/location-based search and update
- Memory overhead of $n$ pointers plus $O(n/b)$ additional storage, provided that bits can be packed in pointers
- Iterator operations in $O(1)$ time, except that operator -- takes $O(b)$ time
Proof by picture

Iterators can be implemented as pointers to list nodes; packing of bits in pointers is again in use.

**Memory overhead:**

\[ n + O(n/b) \text{ words} \]

- \( O(n/b) \) words
- \( O(n/b) \) headers
- \( b \ldots 4b \) elements per list; elements in sorted order

\[ n \text{ list nodes} \]
Regularity requirements

All transformations; requirements for $\mathcal{D}$:

**Referential integrity:** External references must be kept valid at all times

**Location-based access:** In case of multisets, location-based erase must be supported

**Side-effect freeness:** Let $x$, $y$, and $z$ be three consecutive elements in $\mathcal{D}$. It should be possible to replace $y$ with $y'$, without informing $\mathcal{D}$, as long as $x \leq y' \leq z$. 
Regularity requirements (cont.)

Transformations of elementary dictionaries; requirements for $D$:

**Little redundancy:** Each element is stored only once at the place pointed to by its locator

**Freely movable nodes:** Every node knows who points to it so that, when it is moved, it can inform its neighbours

**Constant in-degree:** So that nodes can be moved in constant time

For example, a red-black tree fulfils all the requirements (if we disallow external references to nodes).
Conclusions

• Pointer packing may be a portability hazard

• It is disadvantageous to give raw pointers as locators for external users since such pointers make memory management difficult

• We would like to understand better data structures maintained in garbage-collected memory

• To be done: rigorous practical experiments