Efficient Data Structures
Summer 2014
Paweł Gawrychowski
Mayank Goswami
Patrick Nicholson
About the course: Marking, etc.

- This is a 9 credit point course: 2+2

- Prerequisites: Basic course in data structures
  - You *should* know asymptotic analysis ($O, o, \Theta, \Omega, \omega$)
  - You *should* know about linked lists/balanced trees
  - You *should* know at least one programming language
    - ADA DOC ASM AWK BASH BF C C# C++ 4.3.2 C++ 4.0.0–8 C99 strict CLPS CLOJ LISP sbcl LISP clisp D ERL F# FORT GO HASK ICON ICK JAR JAVA JS LUA NEM NICE NODEJS CAML PAS fpc PAS gpc PDF PERL PERL 6 PHP PIKE PS PRLG PYTH 2.7 PYTH 3.2.3 PYTH 3.2.3 n RUBY SCALA SCM guile SCM qobi SED ST TCL TECS TEXT WSPC

- Marking scheme:
  - 60% exam
  - 30% homework sheets (must get 50% on homework)
  - 10% project (research/survey/implementation)
    - Groups of up to 3 people; more details will follow
We will have weekly homework sheets
  ◦ Each homework sheet will have
    • Theory problems (i.e., proofs)
    • Programming problems (at most 20% of homework)
      • These are to be submitted on SPOJ
      • See homework sheet for details

We will also have weekly tutorials
  ◦ Each tutorial review the previous week’s assignment
  ◦ You must actively participate in the tutorial sessions
About the course: Short Outline

- Models of computation
- Implicit Data Structures (Comparison)
  - Membership (Dictionary) Problem, Multikey Search
- Succinct Data Structures (Word–RAM)
  - Static problems: rank/select, trees, graphs, etc.
  - Cell Probe lower bounds for succinct data structures
  - Discussion of dynamic memory models
- Static predecessor searching (Word–RAM)
- Making data structures dynamic
- Persistence and applications (Pointer Machine/Word–RAM)
- Lower bounds (Comparison, Pointer Machine, Cell–probe, etc.)
- Introduction to the External Memory (I/O) Model
  - classic data structures: B–trees, Buffer trees.
- Efficient data structures in external memory
  - Generalizing word–RAM structures to the I/O model
  - Lower bounds on external memory data structures
Let’s get Philosophical

- Why do we do algorithm analysis?
  - What are the goals?
    - Compare different algorithms
    - Determine which algorithm to use in which case
  - What is the end result of the analysis?
    - Input: an algorithm and some input parameters
      - We want a number: lower better than higher

- How do we do the analysis?
  - Computers are very complicated
  - Instead we analyse simpler *models of computation*
There are *many* different models
- Comparison-based, Word-RAM, Cell-Probe, I/O, Pointer machine, Cache-oblivious, etc.

It is important to understand the limitations
- This helps with understanding practicality
- Models often focus on one particular aspect
- We will discuss cases where it can be misleading

Example: Sorting
IMPLICIT DATA STRUCTURES

Summer 2014
Efficient Data Structures
Patrick Nicholson
Why do we care about space efficiency?

Practical reasons:
- In many computations the limiting factor is memory
- The memory hierarchy
- Saving even a small constant factor in space means big money
- Many computing devices often have less memory resources:
  - Smartphones
  - Microcontrollers
  - Sensors
  - Facebook enabled toaster
Why do we care about space efficiency?

Theoretical Reasons:
- Answer fundamental questions about computation:
  - “How much extra space do we need to answer queries about data?”
  - “Can we compress data and still answer questions about it?”
  - “Which types of queries are impossible to efficiently support?”
  - “Are pointers necessary?”
- It is fun 😊
What is the model?

- Basic Idea: data is stored in an array $A[1..n]$
  - The “structure” consists of the order of the data
  - A “pointer” is just an integer in $A[1..n]$

- Only need to know the value $n$
  - AKA: strict implicit data structure
  - Another option: $O(1)$ extra data allowed

- Only allowed to make comparisons:
  - $a < b, a = b, a > b$

Comments?
Implicit Data Structures

- You probably already know one...

- Heaps perform the following operations:
  - Insert(x): add key x
  - Delete–Min(): delete and return the smallest key
  - Get–Min(): return the smallest key

- Insert(x) and Delete–Min() take $\Theta(\log n)$ time
- Get–Min() takes $\Theta(1)$ time
Binary Heap

- Heap Properties:
  - Complete binary tree except for the last level
  - Each node’s key is at least as small as its children’s
The heap structure is a partial order

- A partial order is a binary relation that is:
  - Reflexive, Antisymmetric, and Transitive
- Think of a directed acyclic graph with/without shortcuts
The heap structure is a partial order
- A partial order is a binary relation that is:
  - Reflexive, Antisymmetric, and Transitive
- Think of a directed acyclic graph with/without shortcuts
The heap structure is a partial order

- A partial order is a binary relation that is:
  - Reflexive,
  - Antisymmetric, and
  - Transitive

Think of a directed acyclic graph with/without shortcuts

A Maximal Chain
The heap structure is a partial order

- A partial order is a binary relation that is:
  - Reflexive, Antisymmetric, and Transitive
- Think of a directed acyclic graph with/without shortcuts

The Maximum Antichain
Let $C$ and $A$ be *maximum* chain and antichain.

**Dilworth’s Lemma**: Given an arbitrary partial order on $n$ elements the product $|C| \times |A| \geq n$

- $|A| = 7, |C| = 4, n = 13$
  - Seems to check out

Remember this for later!
Back to the Binary Heap

- **Heap Embedding:**
  - Left-child of node $i = 2i$
  - Right-child of node $i = 2i + 1$
  - Parent of $i = \lfloor i/2 \rfloor$
Binary Heap

- Insertion
Binary Heap

- Insertion
Binary Heap

- Insertion
Binary Heap

- Insertion
Binary Heap

- Deletion
Binary Heap

- Deletion
Binary Heap

- Deletion
Binary Heap

- Deletion
Beyond the Heap

- What else can be made implicit?

- Toy Problem: Dynamic Membership
  - Design a data structure that can:
    - Insert($x$)
    - Delete($x$)
    - Member($x$)

- Heap doesn’t work well for member
  - Has very large antichains
Beyond the Heap

- Dynamic Membership
  - Insert($x$)
  - Delete($x$)
  - Member($x$)
  - Heap:
    - Insert $\rightarrow \Theta(\log n)$, Delete $\rightarrow \Theta(n)$, Member $\rightarrow \Theta(n)$
  - Unsorted list:
    - Insert $\rightarrow \Theta(1)$, Delete $\rightarrow \Theta(n)$, Member $\rightarrow \Theta(n)$
  - Sorted list:
    - Insert $\rightarrow \Theta(n)$, Delete $\rightarrow \Theta(n)$, Member $\rightarrow \Theta(\log n)$

- What other trade-offs exist?
Beaps: Biparental Heaps

Beap Properties:

- Partitioned into $\sqrt{2n}$ blocks:
  - $i$–th block $[i(i + 1)/2 + 1..i(i + 1)/2]$

- $k$–th element in the $j$–th block is no larger than the $k$–th and $(k + 1)$–th in $(j + 1)$–th block
Searching for 17
Beaps: Biparental Heaps

- Searching for 17
Beaps: Biparental Heaps

- Searching for 4
Beaps: Biparenteral Heaps

- Inserting 1
Beaps: Biparental Heaps

- Inserting 1
Beaps: Biparental Heaps

- Inserting 1
Beaps: Biparental Heaps

- Inserting 1
Beaps: Biparental Heaps

- Inserting 1

![Diagram of a binary tree representing Beaps]

- The tree structure illustrates the arrangement of elements in a heap, with each node having at most two children.
Beaps: Biparental Heaps

- Same idea as binary heap for deletion
- All three operations take $\Theta(\sqrt{n})$ time
- Elements stored in fixed partial order
  - Just as in the heap
Beaps: Biparental Heaps

- **Theorem (Munro and Suwanda 1980):** If an implicit data structure containing \( n \) elements carries no structural information other than a fixed partial order on the stored values, then

\[ U \cdot S \geq n \]

- \( U \leftarrow \) worst case # of data moves during an update
- \( S \leftarrow \) worst case # of comparisons made during a search
But there is an assumption...

Source: XKCD (http://xkcd.com/1339/), Copyright Randall Munroe (2014), Creative Commons Attribution-NonCommercial 2.5 License.
Rotated Lists

- What about non-partial orders?

- A rotated list: \{7, 11, 13, 14, 1, 4, 5, 6\}
  - Not hard to see that it is possible to modify binary search to find the minimum in the list
    - Caveat: (most) of the elements have to be distinct

- We can do better by using rotated lists
  - But we must make the distinctness assumption!
Basic Rotated List Scheme

- **Data structure:**
  - Keep $\sim \sqrt{2n}$ rotated lists, list $i$ is of length $i$.
  - Invariant: Elements in list $i$ are smaller than list $i + 1$.

- **Member:**
  - Find two consecutive blocks that straddle query element
  - Search in the smaller block
  - Total cost: $\Theta(\log n)$

- **Insertion:**
  - Find block, insert
  - Swap max to min for each larger block
  - Total cost: $\Theta(\sqrt{n} \log n)$
Extensions to Rotated Lists

- **Munro and Suwanda (1980):**
  - Combine Beap and Rotated List to get
    - $\Theta(n^{1/3}\log n)$ for each operation

- **Fredrickson (1983):**
  - Applied recursion to Rotated Lists to get
    - $\Theta(\log n)$ time for Member($x$)
    - $\Theta(n^{\sqrt{2}/\log n}\log^{3/2} n)$ time for Insert($x$) and Delete($x$)
Fredrickson’s Rotated Lists

Fredrickson considered blocking schemes:
- Partition the array into $r$ blocks $B(1), ..., B(r)$
- There is a function $f$ s.t. $|B(i)| = f(i)$
- The $j$-th block contains elements $1 + \sum_{i=1}^{j-1} f(i)$ to $\sum_{i=1}^{j} f(i)$
  - The Basic Rotated List Scheme has $f(i) = i$

Data structure idea: *Bootstrapping*
- Sometimes we can plug a data structure into itself
  - Let $D_1$ have $f(i) = i$ and each block be a rotated list
  - Let $D_2$ have $f(i) = i^2$ and each block be $D_1^*$
    - This gives us $\Theta(\log n)$ search, and $\Theta(n^{1.5} \log n)$ updates!
  - Let $D_3$ have ...
Beyond Rotated Lists

- **Theorem (Munro 1986):** There is an implicit data structure for the membership problem that has worst case $\Theta(\log^2 n)$ time for Member($x$), Insert($x$), and Delete($x$)

- What we really want is an balanced search tree
  - So, let's see if we can make such a tree implicit
Theorem (Munro 1986): There is a data structure for the membership problem that occupies $n + k^2$ array locations, and uses an additional $k + \Theta(n/k)$ pointers, counters, and flags. $\text{Member}(x)$ takes $\Theta(\log n)$ time, and $\text{Insert}(x)$ and $\text{Delete}(x)$ take time $\Theta(k + \log n)$ time.

Invariant #1: AVL node stores $k$ consecutive elements
- A node consists of $k$ locations for elements
  - Also a constant number of pointers, flags, and counters
- We take node sized blocks from the end of the data array
We need some extra mechanism to update

Invariant: 0 to k–1 consecutive elements between AVL nodes

The elements between two nodes are called a *maniple*
We keep pointers to $k - 1$ doubly linked lists
- Each linked list will also consist of nodes
- List $i$ will consist of all maniples of $i$ elements
- Each AVL node stores a pointer to its maniple
We keep pointers to $k - 1$ doubly linked lists
- Each linked list will also consist of nodes
- List $i$ will consist of all maniples of $i$ elements
- Each AVL node stores a pointer to its maniple

Each list node may contain maniples for up to $k$ AVL nodes
- This set of AVL nodes is called the cohort of the list node
  - We keep circular linked lists so we can find all AVL nodes in a cohort
    (Yes, there are a lot of pointers!)
Managing Maniples (Updates)

- Memory Management:
  - When we need a new node, get it from the array
    - New list nodes inserted at the head of the list
  - To delete a maniple, swap contents with head
    - Must update maniple/cohort pointers in process
    - If head underflows, swap with final node in array
    - Overall this requires $\Theta(\log n + k)$ time

- Thus, we can assume the following primitives:
  - PromoteManiple($p, i, x$): move maniple pointed to by $p$, of size $i$ into maniple list $i + 1$, while inserting $x$ into the correct position
  - DemoteManiple($p, i, x$): move maniple pointed to by $p$, of size $i$ into maniple list $i - 1$, and delete $x$
Performing Operations

- Insert is conceptually very easy:
  - Two cases: both more or less the same
    - Insert into an AVL node → bump max element into maniple
    - OR Insert directly into maniple
  - So, we what we really need is to handle maniple insertion:
    - If the maniple is empty, make a new one in list 1
    - If the maniple is already of size $k-1$, make AVL node
    - Otherwise, we use PromoteManiple

- Deletion is analogous 😊

- Search:
  - In the AVL tree: $\Theta(\log n)$
  - In a node: $\Theta(\log k)$
  - Total: $\Theta(\log n)$
Recall that nodes store $k$ consecutive values:
- We can encode $k/2$ bits in these values!

\[
\begin{array}{cccccccccccc}
2 & 3 & 4 & 5 & 7 & 12 & 13 & 17 & 18 & 20 & 22 & 29 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
3 & 2 & 4 & 5 & 12 & 7 & 13 & 17 & 20 & 18 & 29 & 22 \\
\end{array}
\]

- Takes $\Theta(k)$ time to decode/encode a pointer!
  - We will set $k = \log n$ and get $\Theta(\log^2 n)$ time for all ops.
We set $k = c \lceil \log n \rceil$, where $c$ is a big constant
  ◦ e.g., $c = 10$ it will be large enough

Dealing with the cruft:
  ◦ There are $k - 1$ linked lists of maniples
    • Each list can have up to $k - 1$ unused locations
    • Thus, we are wasting $\Theta(k^2)$ locations in total!
    • We store these in the final locations of the array
      • Problem solved with extra pointers

Are we done?
Annoying issue:
- The value of $\lceil \log n \rceil$ will change eventually
- Luckily, there is an easy solution:
  - Keep $\Theta(\log \log n)$ copies of the membership structure
    - Structure $i$ stores $2^{2^i}$ elements
    - Perform search/updates on all the dictionaries
      - Similar to the rotate list idea for updates
    - We can maintain the running time of $\Theta(\log^2 n)$

The end?
Several improvements since:

- Franceschini et al. (2004):
  - All operations $\Theta(\log^2 n / \log \log n)$

- Franceschini and Grossi (2003, 2006):
  - All operations $\Theta(\log n)$

- Brodal et al. (2012, 2013)
  - Other desirable properties
Next Problem: Multikey Search

- Unlike the last problem, this one will be static

- **Input:**
  - A set of n records, each record has k keys

- **Goal:**
  - Order records for efficient searching using *any* key
Two Key Case: Attempt #1

- Sort the records according to key #1
- Break it up into blocks of size $\sqrt{n}$
- Sort each block according to key #2

- Search using key #1 takes $\Theta(\sqrt{n})$ time
- Search using key #2 takes $\Theta(\sqrt{n \log n})$ time

- Can we do better?
Two Key Case: Attempt #2

- We store the elements in a BST layout (like the heap)
  - Odd levels: split using key #1
  - Even levels: split using key #2
- What is the running time?
  - \( \Theta(\sqrt{n}) \) for searching under either key
  - If we know \( j \) of \( k \) keys: \( \Theta(\max( n^{1-j/k}, \log n)) \)

This is really a \( kd \)-tree
Kd–trees

- We can also do **orthogonal range reporting**:
  - Time complexity: $\theta(\sqrt{n} + t)$ where $t$ is output size
  - Proof: Consider the number of *cells* that are cut by a horizontal or vertical line...
A Relevant Lower Bound

- Theorem (Alt, Mehlhorn, Munro 1984): Assume all comparisons are required to involve the element for which we are searching. If \( n \) elements can be arranged in an array such that any of \( p \) different permutations of the ascending order may occur, then searching requires \( \Omega(p^{1/n}) \) comparisons.
Consider the following permutation:
\[ \pi = (3,2,0,1,4,6,5) \]
as a directed graph:

- A permutation induces a set of *cycles*
  - The length of a cycle is the number of elements

- A permutation which is its own inverse is called an *involution*
  - In an involution, all cycles are of length \( \leq 2 \)
  - Example: \( \pi = (1,0,3,2,5,4,7,6) \) or the bit encoding trick
Consider the following ordering scheme:

- Take the first $n/4$ odd elements and pair them arbitrarily with the last $n/4$ odd elements
  - This admits $(n/4)!$ permutations

- Lower bound says search time should be $\Omega(n^{1/4})$...
  - But we can still search in $\Theta(\log n)$ time if we make comparisons that don’t involve the query element!
Two Key Case: Attempt #3

- We will use the involution trick to show:

- **Theorem (Munro 1987):** The static two-key search problem is solvable in $\Theta(\log^2 n \log \log n)$ time for searching under either key.
Two Key Case: Attempt #3 (2)

- **Feldman’s scheme:**
  - Elements in position $0 \mod 2$ in sorted order
  - Elements in position $1 \mod 2$ permuted

- **Munro’s 2-key scheme:**
  - Start by sorting by key 1
  - Records in $0 \mod \log n$ sorted by key 1
    - Call these 1-guides
  - Conceptually $\log n - 1$ data structures
    - $D_i$ for records in position $i \mod \log n$
  - **Invariant:** $x \in D_i$ straddled by 1-guides

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>12</td>
<td>14</td>
<td>17</td>
<td>19</td>
<td>22</td>
<td>25</td>
<td>29</td>
<td>31</td>
<td>33</td>
<td>37</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>12</td>
<td>22</td>
<td>46</td>
<td>17</td>
<td>11</td>
<td>13</td>
<td>33</td>
<td>34</td>
<td>37</td>
<td>10</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
For each $D_i$:
- Put first half of the records into second half of array sorted by key 2
For each $D_i$:
  - Put first half of the records into second half of array sorted by key 2
Two Key Case: Attempt #3 (3)

- For each $D_i$:
  - Put first half of the records into second half of array sorted by key 2

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>12</td>
<td>14</td>
<td>17</td>
<td>19</td>
<td>22</td>
<td>25</td>
<td>29</td>
<td>31</td>
<td>33</td>
<td>37</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>12</td>
<td>14</td>
<td>17</td>
<td>19</td>
<td>22</td>
<td>25</td>
<td>29</td>
<td>31</td>
<td>33</td>
<td>37</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>12</td>
<td>22</td>
<td>46</td>
<td>17</td>
<td>11</td>
<td>13</td>
<td>33</td>
<td>34</td>
<td>37</td>
<td>10</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>19</td>
<td>33</td>
<td>5</td>
<td>7</td>
<td>31</td>
<td>22</td>
<td>14</td>
<td>17</td>
<td>3</td>
<td>12</td>
<td>25</td>
<td>29</td>
<td>9</td>
<td>4</td>
<td>37</td>
</tr>
<tr>
<td>9</td>
<td>34</td>
<td>8</td>
<td>22</td>
<td>46</td>
<td>3</td>
<td>37</td>
<td>10</td>
<td>11</td>
<td>10</td>
<td>2</td>
<td>17</td>
<td>12</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Two Key Case: Attempt #3 (3)

- For each $D_i$:
  - Put first half of the records into second half of array sorted by key 2
Keep doing this recursively for each $D_i$:
- Put the first half into the second sorted by key 2
- Put the first quarter into the second sorted by key 2
- Put the first eighth into the second sorted by key 2
- ...
- Stop after $\log \log n + c$ recursive calls for some $c > 0$
- Call the $j$-th sorted chunk from the right \textit{level} $j$

$\Theta \left( \frac{n}{\log^2 n} \right)$ records
We now show how to:
- Search among the 1-guides using key 2
- Search among the unsorted portions of \( D_i \) (either key)

Idea that we have seen before:
- Encode pointers in the pairs of records sorted by key 2
- We have \( \Theta(n) \) such records → can encode \( \Theta(n/\log n) \) pointers
- We can use these pointers to encode search trees
Next: how to search using key 2 on the remaining records

- We have $\Theta(\log n)$ data structures
- Each structure has $\Theta(\log \log n)$ levels
- Each level is sorted using key 2
- Overall time: $\Theta(\log^2 n \log \log n)$
Finally: searching using key 1

- The “much more interesting case”
- **Remember (Invariant):** each $y \in D_i$ is straddled by 1-guides
  - Thus, we can determine where the query element $x$ should be
    - That is, we can find a range $r$ of $\log n$ positions (1 per $D_i$)
- We need to do a binary search within $r$
  - $\Theta(\log \log n)$ to search $r$
  - For each $D_i$ we have to track down the correct record
    - How long does this take?
Tracking down elements

- Consider a single $D_i$
Finally: searching using key 1
- The “much more interesting case”
- **Remember (Invariant):** each \( y \in D_i \) is straddled by 1-guides
  - Thus, we can determine where the query element \( x \) **should** be
    - That is, we can find a range \( r \) of \( \log n \) positions (1 per \( D_i \))
- We need to do a binary search within \( r \)
  - \( \Theta(\log \log n) \) to search \( r \)
  - For each \( D_i \) we have to track down the correct record
    - Tracking down: \( \Theta(\log n) \) moves, each move: \( \Theta(\log n) \) cost

**Overall time:** \( \Theta(\log^2 n \log \log n) \)
These results all generalize to 3 or more keys

Fiat et al. (1988) essentially settled it:
- With $k$ keys we can search in $\Theta(k \log k \log n)$ time
- This solution is somewhat complicated
  - Basic Idea: select guides using Hall’s Theorem