Today: succinct data structures
= small space, often static

**Implicit DS**: space = OPT + \( O(1) \) bits
information theoretic for rounding
- typically, DS is “just the data”, permuted in some order
- e.g. sorted array, heap

**Succinct DS**: space = OPT + o(OPT)
- lead constant of 1

**Compact DS**: space = O(OPT)
- often a factor of \( w \) smaller than “linear-space” data structures
  e.g. suffix trees use \( O(n) \) words for \( n \)-bit string
Minisurvey:

- Implicit dynamic search tree:
  [Franceschini & Grossi - ICALP 2003/WADS 2003]
  $O(lg n)$ worst-case time/insert, delete, predecessor
  also $O(lg_b N)$ cache oblivious

- Succinct dictionary:
  $lg(n) + O(n \frac{lg lg n}{lg n})$ bits
  $O(1)$ membership query (static)
  [Brodinik & Munro - SICOMP 1999; Pagh - SICOMP 2001]

- Succinct binary trie:
  $C_n = (\frac{2^n}{n+1}) \sim 4^n$ such tries
  (Catalan)
  $lg C_n + o(lg C_n) = 2n + o(n)$ bits
  $O(1)$ left child, right child, parent, subtree size
  [Munro & Raman - SICOMP 2001]

- Succinct $k$-ary trie:
  (e.g., suffix tree)
  $C^k_n = (\frac{k^{n+1}}{kn+1})$ tries,
  $lg C^k_n + o$ bits
  $O(1)$ child with label $i$, parent, subtree size...
  [Farzan & Munro - SWAT 2008]

- Succinct permutations:
  [Munro, Raman, Raman, Rao - Algorithmica 2005]
  $lg n! + o(n)$ bits, $O(\frac{lg n}{lg lg n})$ time to compute $\pi^k(x)$ & $k$
  $(1+\epsilon) n lg n$ bits, $O(1)$ time $\pi^k$
  (including $k<0$)

- Generalizes to functions
  [Munro & Rao - ICALP 2004]

- Compact Abelian groups:
  [Farzan & Munro - ISSAC 2006]
  $O(lg n)$ bits for group of order $n$ (!) or elt. in group
  $O(1)$ multiply, inverse, equality testing

- Graphs
  [Farzan & Munro - ESA 2008; Barbay, Aleardi, He, Munro - ISAAC 2007]
Level-order representation of binary tries: [Munro]

for each node in level order:
- write 0/1 for whether have left child
- write 0/1 for whether have right child

⇒ 2n bits

e.g:

```
A
  B
  G
  D
C
  F
```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(1) 1 1 0 1 1 1 0 1 0 0 0 0 0 0
A B C * D E F * G * * * * * * *

Equivalently:
- append external node (*) for each missing child
- for each node in level order:
  - write 0 if external, 1 if internal

⇒ extra leading 1 (2n+1 bits)
Navigation: (in external-node view)
left & right children of $i$th internal node
are at positions $2i$ & $2i+1$

Proof: [possible much simpler proof
by induction on $i$]
- suppose $i$th internal node is at position $i+j$
i.e. $j$ external nodes up to $i$th internal node
- $i-1$ previous internal nodes have
  $2(i-1)$ children
- $i-1$ already seen as internal nodes (all but root)
- $j$ already seen as external nodes
$\Rightarrow 2(i-1)-(i-1)-j = i-j-1$ left intervening
$\Rightarrow$ left child at pos. $(i+j)+(i+j-1)+1 = 2i$ \[ \square \]

Rank & Select in bit string:
$\text{rank}_1(i) = \#1$'s at or before position $i$
$\text{select}_1(j) = \text{position of } j$th $1$ bit

$\Rightarrow \text{left-child}(i) = 2 \text{rank}_1(i)$
$\text{right-child}(i) = 2 \text{rank}_1(i)+1$
$\text{parent}(i) = \text{select}(\lceil i/2 \rceil)$

(but subtree-size impossible in level-order rep)
Rank: [Jacobsen - FOCS 1989]

1. Use lookup table for bitstrings of length $\frac{1}{3} \lg n$.
   \[\Rightarrow O(\sqrt{n \lg n \lg \lg n})\] bits of space.

   - Bitstring query answer.

2. Split into $(\lg^2 n)$-bit chunks:

   \[\frac{\lg^3 n}{\lg^2 n}\]

   - Store cumulative rank: $\lg n$ bits.
   \[\Rightarrow O\left(\frac{n}{\lg^2 n \lg n}\right) = O\left(\frac{n}{\lg n}\right)\] bits.

   - Couldn't afford $\lg n$-bit chunks.

3. Split each chunk into $(\frac{1}{2} \lg n)$-bit subchunks:

   \[\frac{\lg n}{\frac{1}{2} \lg n}\]

   - Store cumulative rank within chunk: $\lg \lg n$ bits.
   \[\Rightarrow O\left(\frac{n}{\lg n \lg \lg n}\right) = o(n)\] bits.

4. Rank = rank of chunk
   + Relative rank of subchunk within chunk
   + Relative rank of element within subchunk (via lookup table).

   \[\Rightarrow O(1)\] time,

   \[O\left(n \frac{\log \log n}{\log n}\right)\] bits.

- $O(n / \lg^k n)$ bits possible for any $k = O(1)$.

[Pătraşcu - FOCS 2008]
Select: [Clark & Munro – Clark's PhD 1996]

1. store array of indices of every \((\log n \log \log n)\)th 1 bit
   \[ \Rightarrow O(\frac{n}{\log n \log \log n}) = O(\sqrt{\frac{n}{\log n}}) \] bits

2. within group of \(\log n \log \log n\) 1 bits, say \(r\) bits:
   if \(r \geq (\log n \log \log n)^3\)
   then store array of indices of 1 bits in group
   \[ \Rightarrow O(\frac{n}{\log n \log \log n}) (\log n \log \log n) \log n \]
   "such groups" "1 bits" "index"
   else reduced to bitstring of length \(r \leq (\log n \log \log n)^3\)

3. repeat 1 & 2 on all reduced bitstrings
   to reduce to bitstrings of length \((\log \log n)^{(1)}\)

1. store relative index \((\log \log n)\) bits) of every
   \((\log \log n)^2\)th 1 bit \((\log \log n \log \log n \log \log n \log \log n\) also OK but bigger
   \[ \Rightarrow O(\frac{n}{(\log \log n)^2 \log \log n}) = O(\frac{n}{\log \log n}) \] bits

2. within group of \((\log \log n)^2\) 1 bits, say \(r\) bits:
   if \(r \geq (\log \log n)^4\)
   then store relative indices of 1 bits
   \[ \Rightarrow O(\frac{n}{(\log \log n)^3} (\log \log n) \log \log n) \]
   "such groups" "1 bits" "rel. index"
   else reduced to bitstring of length \(r \leq (\log \log n)^4\)

4. use lookup table for bitstrings of length \(\leq \frac{1}{2} \log n\)
   \[ \Rightarrow O(\sqrt{n} \log n \log \log n) \]
   "bitstrings" "query" "answer"

\[ \Rightarrow O(1) \] query, \(O(\frac{n}{\log \log n})\) bits
- \(O(n^{1/2k} \log n)\) bits \(\forall k = O(1)\) [Patrascu - FOCS 2008]
Binary tries as balanced parentheses: [Munro & Raman - SICOMP 2001]

- balanced parens (= bitstring)
  \[(((()))))((()))\]
  \*A B B C C D D A E F F E G G*

- node
- left child
- right child
- parent
- first child
- next sibling
- prev. sibling
- or parent
- subtree size
  - size(node) +
  - sizes(right siblings)

- similar to (& using) rank & select, can find matching & enclosing parens in O(1) time, O(n) space

  \[\Rightarrow\text{all operations above in } O(1) \text{ time} \]

- from subtree size can accumulate index of node for auxiliary data (e.g. pointer to text)