Today: succinct data structures I (of 2)
- Survey
- succinct binary tries
  - level-order
  - via balanced parentheses
- succinct rank & select

Goal: 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:
  \[O(\log n)\] worst-case time/insert/delete/predecessor
  also \(O(\log_b N)\) cache oblivious

- succinct dictionary:
  \(\frac{\log (n^4) + O(n \log \frac{\log n}{\log n})}{\log n}\) bits
  \(O(1)\) membership query (static)

- succinct binary trie:
  \(C_n = \binom{2n}{n}/(n+1) \approx 4^n\) such tries (Catalan)
  \(\log C_n + o(\log C_n) = 2n + o(n)\) bits
  \(O(1)\) left child, right child, parent, subtree size
  \(- O(1)\) ins/del. leaf, subdivide edge

- succinct k-ary trie: (e.g. suffix tree)
  \(C_k^n = \binom{k(n+1)}{n}/(kn+1)\) tries, \(\log C_k^n + o\) bits
  \(O(1)\) child with label \(i\), parent, subtree size, ...

Improving

- succinct permutations:
  \(\log n! + o(n)\) bits, \(O(\frac{\log n}{\log \log n})\) time to compute \(\pi^k(x) \land k\)
  \((1+\epsilon) n \log n\) bits, \(O(1)\) time \(\pi^k\) (including \(k<0\))

Generalizes to functions

- compact Abelian groups:
  \(O(\log n)\) bits for group of order \(n (!)\) or elt. in group
  \(O(1)\) multiply, inverse, equality testing

- graphs \([Farzan & Munro - ESA 2008; Barbay, Aleardi, He, Munro - Alg. 2012]\)
- implicit n-bit ints: inc./dec. in \(O(\log n)\) bit reads & \(O(1)\) bit writes

\(\text{OPEN: } O(1)\) word RAM?
**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

\[ \Rightarrow 2n \text{ bits} \]

**Example:**

```
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

\[ \Rightarrow \text{extra leading 1} \quad (2n+1 \text{ bits}) \]
Navigation: (in external-node view)
left & right children of ith internal node are at positions 2i & 2i+1

Proof: by induction on i:
- just after (i-1)st internal node's children (as external nodes have no children)
  - either same level or new

Rank & Select in bit string:
rank_1(i) = # 1's at or before position i
select_1(j) = position of jth 1 bit

⇒ left-child(i) = 2 \cdot rank_1(i)
right-child(i) = 2 \cdot rank_1(i) + 1
parent(i) = select(\lfloor i/2 \rfloor)

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

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

2. Split into \((\lg^2 n)\)-bit chunks:
   \[ \frac{\lg^2 n}{\lg^3 n} \]
   Store cumulative rank: \(\lg n\) bits
   \[ \Rightarrow O\left(\frac{n}{\lg^2 n} \lg n\right) = O\left(\frac{n}{\lg n}\right) \text{ bits} \]
   (couldn’t afford \(\lg n\)-bit chunks)

3. Split each chunk into \((\frac{1}{3} \lg n)\)-bit subchunks:
   \[ \frac{\frac{1}{3} \lg n}{\frac{1}{3} \lg n} \]
   Store cumulative rank within chunk: \(\lg \lg n\) bits
   \[ \Rightarrow O\left(\frac{n}{\lg n} \lg \lg n\right) = o(n) \text{ bits} \]

4. Rank = rank of chunk
   + relative rank of subchunk within chunk
   + relative rank of element within subchunk
   (via lookup table)
   \[ \Rightarrow O(1) \text{ time, } O(n \frac{\log^2 n}{\log n}) \text{ bits} \]

- \(O(n/\lg^k n)\) bits possible for any \(k=O(1)\) [Pătraşcu – FOCS 2008]

- \(O(\frac{\log n}{\log \log n})\) insert/delete/rank/select [He & Munro – SPIRE 2010]
Select: [Clark & Munro – Clark's PhD 1996]

1. Store array of indices of every \((\log n \log \log n)\)th 1 bit
   \[
   \Rightarrow O\left(\frac{n}{\log n \log \log n} \log n\right) = O\left(\frac{n}{\log n}\right) \text{ bits}
   \]

2. Within group of \(\log n \log \log n\) 1 bits, say \(r\) bits:
   
   if \(r \geq (\log n \log \log n)^2\)
   
   then store array of indices of 1 bits in group
   \[
   \Rightarrow O\left(\frac{n}{(\log n \log \log n)^2} \left(\log n \log \log n\right) \log n\right) = O\left(\frac{n}{\log n}\right) \text{ bits}
   \]

   # such groups # 1 bits index

   else reduced to bitstring of length \(r \leq (\log n \log \log n)^4\)

3. Repeat 1 & 2 on all reduced bitstrings to reduce to bitstrings of length \((\log \log n)^9(1)\)

   1. Store relative index \((\log \log n)\) bits of every \((\log \log n)^4\)th 1 bit \((\log \log n \log \log n\) also OK but bigger)
   \[
   \Rightarrow O\left(\frac{n}{(\log \log n)^4} \log \log n\right) = O\left(\frac{n}{\log \log n}\right) \text{ bits}
   \]

   2. Within group of \((\log \log n)^4\) 1 bits, say \(r\) bits:
   
   if \(r \geq (\log \log n)^8\)
   
   then store relative indices of 1 bits
   \[
   \Rightarrow O\left(\frac{n}{(\log \log n)^8} \left(\log \log n\right)^3 \log \log n\right) = O\left(\frac{n}{\log \log n}\right) \text{ bits}
   \]

   # such groups # 1 bits rel. index

   else reduced to bitstring of length \(r \leq (\log \log n)^{24}\)

4. Use lookup table for bitstrings of length \(\leq \frac{1}{2} \log n\)
   \[
   \Rightarrow O\left(\sqrt{n} \log n \log \log n\right)
   \]

   # bitstrings query answer

   \[
   \Rightarrow O(1) \text{ query, } O\left(\frac{n}{\log \log n}\right) \text{ bits}
   \]

   \[
   O\left(n^{\frac{1}{k}} \log n\right) \text{ bits } \forall k = O(1) \quad [Patrascu - FOCS 2008]
   \]
Binary tries as balanced parentheses: [Munro & Raman - STOCMP 2001]

binary trie       ordered tree       balanced parens (bitstring)

A     E     *     A   E   G
B     C     D     B   C   D
                   F     F

node
left child
right child
parent
subtree size
leaf

left paren. [& matching right]
first child
next sibling
next char. [if else none]
prev. sibling or parent
char. after matching [if ]
prev. char. ) ⇒ its matching ( prev. char. ( ⇒ that ( ½distance to enclosing )

size(node) + sizes(right siblings)
leaf without right sibling

# leaves in Subtree

rank(()) of enclosing
-rank(()) of here

- similar to (& using) rank & select, can find
  matching & enclosing parens. in O(1) time, O(n) space
⇒ all operations above in O(1) time
- from subtree size can accumulate index of node
  for auxiliary data (e.g. pointer to text)