

ADVANCED

overall tree
= binary tree of mini trees

mini trees

micro trees

actual nodes

$\frac{1}{7} \lg n$ nodes

$\lg n$ nodes

n nodes



DATA STRUCTURES

6

Succinct Data Structures

Advanced Data Structures · Summer 2026

Prof. Dr. Sebastian Wild

6 Succinct Data Structures

- 6.1 A Motivating Example
- 6.2 Bitvectors
- 6.3 Supporting Rank
- 6.4 Supporting Select
- 6.5 Compressed Bitvectors
- 6.6 Sequences
- 6.7 Trees
- 6.8 Graphs

6.1 A Motivating Example

Computing over compressed data

- ▶ traditionally:
 - ▶ compression for **archiving** data, minimize size of representation
 - ▶ computation/analysis: minimize time; use extra data structures
 - ↪ always decompress data first!
 - ▶ reaches limits of fast memory for large datasets
- ▶ Approach in space-efficient data structures:
 - ▶ Represent data in compressed form
 - ▶ Augment with **small** index data structures to enable fast queries **directly on compressed representation**
 - ▶ *succinct* = $(1 + o(1)) \cdot$ information-theoretic lower bound

Computing over compressed data

$T[0..n) =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	b	a	n	a	n	a	a	n	d	a	n	a	p	p	l	e

$C[0..m) =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	
0	1	1	1	1	0	0	0	1	0	0	1	0	0	0	1	0	1	1	1	0	1	0	1	0	0	1	1	0	1	1	0	1	1	1	1	1	1	1	1	1	0

Huffman Code

a \mapsto 0
b \mapsto 11100
d \mapsto 11101
e \mapsto 11110
l \mapsto 11111
n \mapsto 10
p \mapsto 110

- ▶ Can decode from beginning \rightsquigarrow fine for storage / transmission



But how to read $T[9]$ without full decoding? How to know where its codeword starts?

- ▶ We don't. But we can store it!
 - ▶ Naive way: Store starting index for each char
 $\rightsquigarrow n$ numbers in $[n]$ $\rightsquigarrow n \lg n$ bits.

Much more than the (compressed) text!



entropy-compressed bitvector
Clever **succinct index** supports random access in constant time with $o(n)$ extra bits!

Dynamic Bitvectors

Recall: Dynamic rank/select lower bounds (Fredman-Saks)

Note: Can use bitvector to represent n numbers from universe $[0..u)$

↪ Cannot have dynamic bitvectors **and** fast rank/select queries, $\Omega\left(\frac{\lg n}{\lg \lg n}\right)$.

↪ *Focus here on static bitvectors.*

6.2 Bitvectors

Bitvector Representation in C++

- ▶ `std::vector<char/int>, bool[]`
 - ▶ easy to access
 - ▶ fastest read and write access
 - ▶ ≥ 1 **byte** per bit
 - ▶ `std::vector<bool>`
 - ▶ special overload
 - ▶ *may* use only 1 bit each
 - ▶ (with my GNU g++ it used same space as `vector<char>`)
 - ▶ `std::vector<uint64_t>`
 - ▶ store 64 bits in one 8 byte integer
 - ▶ use bit tricks to access and write individual bits
- ↪ in practice: `sdsl::bit_vector` (based on packing bits to words)
- ▶ from external library SDSL Lite
 - ▶ <https://github.com/simongog/sdsl-lite>

Rank & Select on Bitvectors

- ▶ $B[0..n]$ static array of n bits (Boolean array).
 - ▶ easy to store using n bits of space
- ▶ $B.\text{rank}(i) = \# \text{ 1s in } B[0..i]$ (first $i \in [0..n]$ positions) (=prefix sum)
- ▶ $B.\text{select}_1(r) = \min\{j : B.\text{rank}(j+1) \geq r\} \cup \{n\}$
= index of r th 1 in B , ($r = 1, 2, \dots$)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
0 1 0 1 0 0 0 0 0 0 1 1 0 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 0 0 0

- ▶ $\text{rank}(12) = \text{rank}(13) = 4$
- ▶ $\text{select}_1(3) = 10$
- ▶ $\text{select}_1(4) = 11$

$\rightsquigarrow B[i] = B.\text{rank}(i+1) - B.\text{rank}(i)$ (no need to discuss access separately)

Goal: Support rank and select on bitvector using $n + o(n)$ bits of space.

Versatile Bitvectors

► Set of integers

► $S \subset [0..u) \rightsquigarrow B[0..u)$ with $B[x] = 1 \iff x \in S$

$\rightsquigarrow B.\text{select}(B.\text{rank}(x + 1)) = S.\text{predecessor}(x)$

\rightsquigarrow efficient intersection, union, etc. via bitwise logical operations

► used, e. g., for “**Roaring Bitmaps**”: one bitvector / integer set per attribute

► Unary encoded integers

► integers $x_1, \dots, x_k \in \mathbb{N}_{\geq 0} \rightsquigarrow B = 1^{x_1}0 1^{x_2}0 \dots 1^{x_k}0$

► $x_1 + \dots + x_j = B.\text{rank}(B.\text{select}_0(j))$

► x_j as difference of prefix sums

► Sparse array index

► suppose $A[0..n)$ is mostly 0, but with some k indices having non-zero values

\rightsquigarrow Store in $V[0..k)$ all non-zero values compactly, in $B[0..n)$ store $B[i] = 1 \iff A[i] \neq 0$

\rightsquigarrow Simulate $A[x] = \begin{cases} 0 & \text{if } B[x] = 0 \\ V[B.\text{rank}[x]] & \text{else} \end{cases}$

► **Note:** In all these cases, rather natural to expect skewed frequencies of 1s and 0s.

6.3 Supporting Rank

Rank – Simple Ideas

- ▶ can answer rank by linear scan ... $O(n)$ time
- ▶ can maintain a *segment tree* with subrange sums
 - ↪ $O(\lg n)$ query time, but also $n \lg n$ bits of extra space!
 - ▶ can support updates (bit flips)
- ▶ can store all answers ↪ $O(1)$ time, but $n \lg n$ extra space
- ▶ hybrid: **subsampling segment tree**
 - ▶ store subrange sums only for blocks of $\lg n$ bits
 - ▶ within block use linear scan
 - ▶ entire blocks covered by segment tree ↪ $O(\log n)$ query time
 - ▶ space: $\Theta\left(\frac{n}{\lg n} \cdot \lg n\right) = \Theta(n)$ bits of extra space
 - ↪ a bit better, but still neither succinct, nor constant time!
(but dynamic!)
 - ▶ Could apply this on larger block sizes and recursively to reduce space, but time remains.

How can we do better?

Rank Index for Bitvector

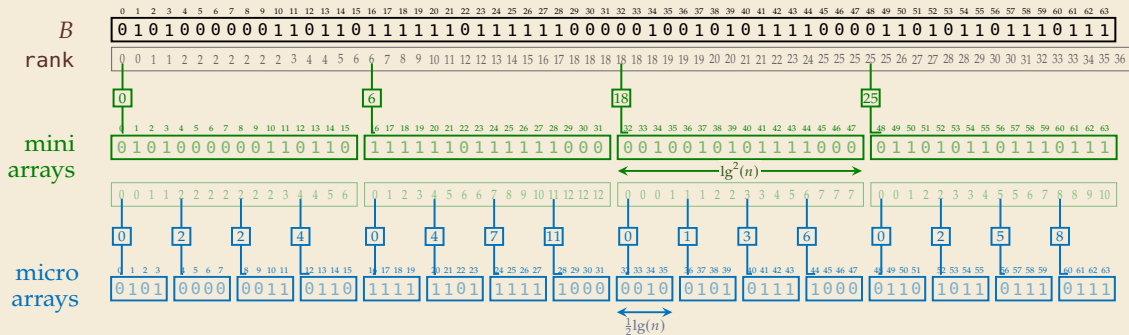
► Apart from B , we store:

1. Rank of first pos. of each **mini array** of $L = \lg^2(n)$ bits $\rightsquigarrow \frac{n}{L} \cdot \lg(n) = \frac{n}{\lg n} = o(n)$ bits

2. Rank of first pos. in **micro array** of $\ell = \frac{1}{2} \lg(n)$ bits *relative to its mini array*

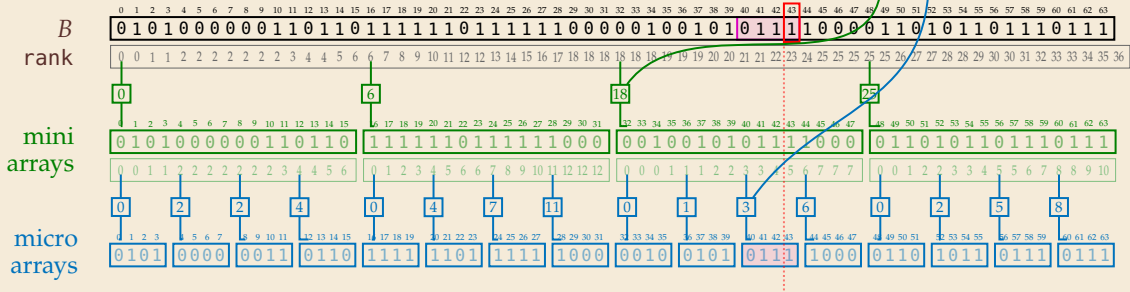
\rightsquigarrow ranks are numbers in $[0..L]$ $\rightsquigarrow \lg L = 2 \lg \lg n$ bits suffice for each

$\rightsquigarrow \frac{n}{\ell} \lg L = \frac{n}{\frac{1}{2} \lg n} \cdot 2 \lg \lg n = \frac{4n \lg \lg n}{\lg n} = o(n)$ bits



How to find rank

$$B.\text{rank}(43) = 18 + 3 + 2 = 23$$



► How to compute $B.\text{rank}(i)$?

- find rank up to element's mini array
- add rank up to element's micro array (mini-array local)
- add micro-array-local rank of position
 - can either do this naively by scanning $\frac{1}{2} \lg n$ bits $\rightsquigarrow O(\log n)$ time
 - or use bit marks and pop-count instructions on CPUs $\rightsquigarrow O(1)$ time
 - or use exhaustive *lookup table!* $\frac{1}{2} \lg n$ bits \rightsquigarrow only $2^{1/2 \lg n} = \sqrt{n}$ different micro arrays

Exhaustive Tabulation

Classic word-RAM technique: bootstrap on small cases with *exhaustive tabulation*

- ▶ **Idea 1:** on word-RAM, can use integers (and hence small bit sequences) as array index
 - ▶ **Idea 2:** for micro subproblems on tiny bit sequences, cannot have many *different* micro subproblems
- ↪ Precompute all micro subproblems in a lookup table, indexed by the subproblem's bitpattern
- ↪ $O(1)$ time for micro subproblem (after preprocessing)

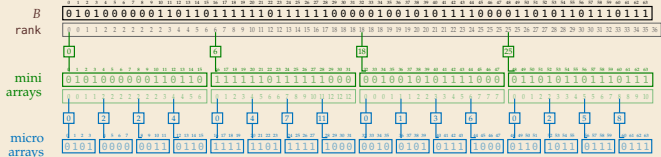
Back to Rank!

Recall: Our task is to compute the local rank in a micro array of $\ell = \frac{1}{2} \lg n$ bits.

Preprocessing: Build table for all possible queries for all *possible* micro arrays

$$\mu R[0..2^\ell][0..\ell] =$$

idx	micro array	rank(0)	rank(1)	rank(2)	rank(3)	rank(4)
0	0000	0	0	0	0	0
1	0001	0	0	0	0	1
2	0010	0	0	0	1	1
3	0011	0	0	0	1	2
4	0100	0	0	1	1	1
5	0101	0	0	1	1	2
6	0110	0	0	1	2	2
7	0111	0	0	1	2	3
8	1000	0	1	1	1	1
9	1001	0	1	1	1	2
10	1010	0	1	1	2	2
11	1011	0	1	1	2	3
12	1100	0	1	2	2	2
13	1101	0	1	2	2	3
14	1110	0	1	3	3	3
15	1111	0	1	2	3	4



- ▶ Note: $\# \text{micro arrays} = \frac{n}{\frac{1}{2} \lg n} \gg$
 $\# \text{different micro arrays} = 2^{1/2 \lg n} = \sqrt{n}$
- ▶ To access micro-array $\text{rank}(i)$
 1. Read micro array *type* (bits), here from B
 2. Treat *type* as binary number $\rightsquigarrow \text{idx } t$
 3. Return $\mu R[t][i]$

Space: $2^\ell \cdot \ell \cdot \lg \ell = \sqrt{n} \cdot \text{polylog}(n) = o(n)$

Succinct Rank – Result

We have shown:

Theorem 6.1 (Succinct bitvector with rank)

Given a bitvector $B[0..n)$, we can construct in $O(n)$ time an index data structure using

$O\left(n \frac{\lg \lg n}{\lg n}\right) = o(n)$ bits of space that can answer $B.\text{rank}(i)$ queries in $O(1)$.

The index requires oracle access to B at query time, i. e., in $O(1)$ time we must be able to read $\Theta(\log n)$ consecutive bits of B . ◀

Four Russians?

The *exhaustive-tabulation technique* is often called “Four Russians trick” in the literature . . .

- ▶ The algorithmic technique was published 1970 by V. L. Arlazarov, E. A. Dinitz, M. A. Kronrod, and I. A. Faradžev



Arlazarov, Dinitz, Kronrod, Faradžev: *On economical construction of the transitive closure of a directed graph*, Dokl. Akad. Nauk SSSR 1970

inofficial translation: <https://minorfree.github.io/FourRussian/>

- ▶ all worked in Moscow at that time . . . but Soviet \neq Russian; Dinitz emigrated to Isreal (Arlazarov and Kronrod are Russian)
- ▶ American authors coined the othering term “Method of Four Russians” . . . name in widespread use

The Original Application: Boolean Matrix Multiplication

Suppose we want to multiply two $n \times n$ Boolean matrices $C = A \cdot B$.

We divide A , B , and C into $\ell \times \ell$ *micro matrices*.

↪ C consists of $\left(\frac{n}{\ell}\right)^2$ micro matrices, each of which is the sum of $\frac{n}{\ell}$ micro-matrix products.

The number of *different* possible micro matrix products is $L = 2^{\ell^2} \cdot 2^{\ell^2}$.

If we pick $\ell = \frac{1}{4}\sqrt{\lg n}$, we have only $L = 2^{2\ell^2} = \sqrt{n}$ different products.

↪ **Exhaustive Tabulation:** Can *precompute* all \sqrt{n} possible micro-matrix sums/products!

For two micro matrices a and b , we store $a \cdot b$ at the offset $a_{1,1} \dots a_{\ell,\ell} b_{1,1} \dots b_{\ell,\ell}$, where we interpret this bitstring as a binary number.

On a word RAM, we can use this as indirect memory access in $O(1)$ time.

↪ Any micro matrix sum/product takes $O(1)$ time after a total of $O(\sqrt{n} \cdot \log^{3/2} n)$ preprocessing.

The total time to compute one micro matrix in C is thus $O\left(\frac{n}{\ell}\right)$.

So the total time to compute C is $O(n^3/\ell^3) = O(n^3/\log^{3/2} n)$.

Note: By taking $n \times \ell$ resp. $\ell \times n$ “micro strips” instead of squares, we can choose $\ell = \Theta(\log n)$ and obtain final time $O(n^3/\log^2 n)$.

6.4 Supporting Select

Select Indices – Simple Ideas

Recall: $B.\text{select}_1(r) = \min\{j : B.\text{rank}(j+1) \geq r\} \cup \{n\}$
 $=$ index of r th 1 in B , ($r = 1, 2, \dots$)

\rightsquigarrow in $O(\log n)$ calls to rank, we can answer select without extra space

▶ Not a terrible option if we only do few select queries

▶ Trivial $O(1)$ solution: store all answers

$\rightsquigarrow m$ indices if $B[0..n)$ contains m 1s $\rightsquigarrow m \lg n$ bits

▶ can be sensible solution if m is small

▶ hybrid solution by subsampling:

▶ explicitly store every k th select answer, binary search in between

$\rightsquigarrow \frac{m}{k} \lg n$ bits

▶ worst case of $O(\log n)$ remains

Can we do better?

Select Index

Theorem 6.2 (Succinct bitvector with select)

Given a bitvector $B[0..n)$, we can construct in $O(n)$ time an index data structure using

$O\left(n \frac{\lg \lg n}{\sqrt{\lg n}}\right) = o(n)$ bits of space that can answer $B.\text{select}(r)$ queries in $O(1)$.

The index requires oracle access to B at query time, i. e., in $O(1)$ time we must be able to read $\Theta(\log n)$ consecutive bits of B . ◀

6.5 Compressed Bitvectors

Recall: Bitvector Applications

- ▶ Many applications of bitvectors naturally yield *unbalanced* bitvectors:
 - ▶ characteristic vector of set $S \subseteq [0..u)$ \rightsquigarrow n ones among u bits
 - ▶ prefix sums via unary encoding \rightsquigarrow #zeros = #numbers; #ones = total sum
 - ▶ sparse array index \rightsquigarrow only useful if many fewer ones than entries

Consider $B[0..n)$ with m 1-bits

- ▶ w.l.o.g. assume $m \leq \frac{n}{2}$ (other case symmetric)
- ▶ Number of different bitvectors $B[0..n)$ with m ones: $\binom{n}{m}$

\rightsquigarrow Should only use $\lg \binom{n}{m} \leq m \lg \left(\frac{n}{m} \right) + 1.4427m$ bits

Coding Tricks

Succinct Compressed Bitvectors

Theorem 6.3

does not need B at query time



Let $B[0..n)$ be a bitvector containing m one bits. In $O(n)$ time we can compute an encoding data structure using

$$\lg \binom{n}{m} + O\left(\frac{n \lg \lg n}{\lg^{1/2} n}\right)$$

bits that supports the following operations in $O(1)$ time:

1. $B[i]$: return the bit at index i
2. $B.\text{rank}_\alpha(i)$ for $\alpha \in \{0, 1\}$
3. $B.\text{select}_\alpha(r)$ for $\alpha \in \{0, 1\}$



Predecessor Strikes Again

Overview: Monotone Minimal Perfect Hashing

- ▶ A function $h : S \rightarrow [m]$ is a *monoton minimal perfect hash function (MMPHF)* if
 - ▶ $h(x) \neq h(y)$ for all $x, y \in S$ (perfect)
 - ▶ $|S| = m$ (minimal)
 - ▶ $h(x) < h(y)$ iff $x < y$ for all $x, y \in S$ (monotone)

$\rightsquigarrow h(x) = S.\text{rank}(x)$ for all $x \in S$

Note that $h(z)$ is completely arbitrary when $z \notin S$

- ▶ A compressed bitvector $B[0..u)$ with $m = |S|$ ones yields a MMPHF via $h(x) = B.\text{rank}(x)$
- ▶ Amazingly, MMPHF exist with $O(n \lg \lg \lg u)$ bits of space (using very different construction)



Belazzougui, Boldi, Pagh, Vigna: *Monotone Minimal Perfect Hashing: Searching a Sorted Table with $O(1)$ Accesses*, SODA 2009

Elias-Fano Encoding

6.6 Sequences

Rank & Select on Sequences

Sequences = strings over Σ

- ▶ Represented as array $T[0..n)$

As for bitvectors, we want to support rank/select queries #occurrences of c

- ▶ $\text{rank}_c(T[0..n), i) = |T[0..i]|_c$
= # c in first i characters of T
- ▶ $\text{select}_c(T[0..n), r)$
= $\min\{j : |T[0..j]|_c \geq r\} \cup \{n\}$
= index of r th c in T , ($r = 1, 2, \dots$)

$T[0..9)$

0	1	2	3	4	5	6	7	8
b	a	n	a	n	a	b	a	n

$\text{rank}_a(T, i)$

0	1	2	3	4	5	6	7	8	9
0	0	1	1	2	2	3	3	4	4

$\text{rank}_b(T, i)$

0	1	2	3	4	5	6	7	8	9
0	1	1	1	1	1	1	2	2	2

$\text{rank}_n(T, i)$

0	1	2	3	4	5	6	7	8	9
0	0	0	1	1	2	2	2	2	3

$\text{select}_a(T, r)$

/	1	3	5	7	9	9	9	9	9
---	---	---	---	---	---	---	---	---	---

$\text{select}_b(T, r)$

/	0	6	9	9	9	9	9	9	9
---	---	---	---	---	---	---	---	---	---

$\text{select}_n(T, r)$

/	2	4	8	9	9	9	9	9	9
---	---	---	---	---	---	---	---	---	---

Rank/select on Sequences again a versatile tool

- ▶ bioinformatics: compressed text indexes, e. g., FM Index
- ▶ Representing graphs

Wavelet Trees

The **Wavelet Trees** for a $T \in [0..n)$ over $\Sigma = [0..\sigma)$

- ▶ supports access to $T[i]$ in $O(\log \sigma)$ time,
- ▶ $\text{rank}_c(T, i)$ and $\text{select}_c(T, r)$ in $O(\log \sigma)$ time, and
- ▶ occupies $\sim n \lg \sigma$ bits of space.



Compressed Wavelet Trees

6.7 Trees

Ordinal Trees

- ▶ rooted tree with unbounded degrees
- ▶ children are ordered (left to right)
- ▶ store only shape of tree (no node labels)
(store labels separately!)

Example uses

- ▶ structure of XML documents
- ▶ MSTs for routing
- ▶ representing parse trees of grammars (grammar-based compression)

Lower Bound

► **Fact:** The number of ordinal trees on $n + 1$ nodes is $C_n = \frac{1}{n+1} \binom{2n}{n}$

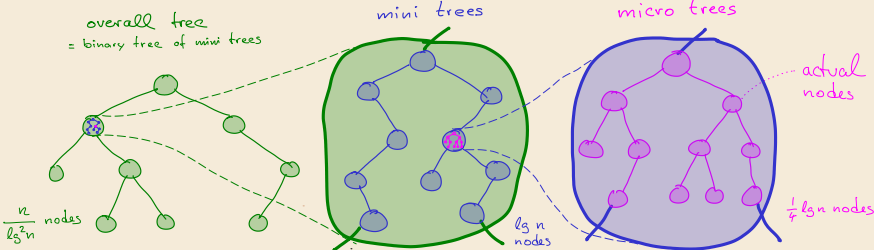
↪ To represent an ordinal tree, we need $\lg C_n \sim 2n$ bits of space

Operations on Trees

Operation	Meaning
<code>root()</code>	The root node
<code>fchild(v)</code>	The first child of node v , if it exists
<code>lchild(v)</code>	The last child of node v , if it exists
<code>nsibling(v)</code>	The next sibling of node v , if it exists
<code>psibling(v)</code>	The previous sibling of node v , if it exists
<code>parent(v)</code>	The parent of node v , if it exists
<code>isleaf(v)</code>	Whether v is a leaf node
<code>nodemap(v)</code>	A unique identifier in $[1, n]$ for v , called its <i>index</i>
<code>nodeselect(i)</code>	The node v with index i
<code>depth(v)</code>	Depth of node v ; the depth of the root is 1
<code>height(v)</code>	Height of node v ; the height of a leaf is 0
<code>deepestnode(v)</code>	A deepest node in the subtree of node v
<code>subtree(v)</code>	Number of nodes in the subtree of v , counting v
<code>isancestor(u, v)</code>	Whether u is an ancestor of v (or v itself)
<code>levelancestor(v, d)</code>	The ancestor at distance d from v ($d = 1$ is <code>parent(v)</code>)
<code>preorder(v)</code>	Preorder value of v (first v , then each subtree)
<code>preorderselect(i)</code>	Node v with preorder i
<code>postorder(v)</code>	Postorder value of v (first each subtree, then v)
<code>postorderselect(i)</code>	Node v with postorder i
<code>children(v)</code>	Number of children of node v
<code>child(v, t)</code>	The t th child of node v , if it exists
<code>childrank(v)</code>	The t such that node v is the t th child of its parent
<code>lca(u, v)</code>	The lowest common ancestor of nodes u and v
<code>leafnum(v)</code>	The number of leaves in the subtree of node v
<code>leafrank(v)</code>	The number of leaves to the left of node v , plus 1
<code>leafselect(i)</code>	The i th left-to-right leaf node in the tree
<code>childrenlabeled(v, l)</code>	Number of children of node v via edges labeled l
<code>labeledchild(v, l, t)</code>	The t th left-to-right child of v by an edge labeled l
<code>childlabel(v)</code>	The label of the edge that leads to node v

LOUDS

Tree Covering



6.8 Graphs

Adjacency Strings