Advanced Data Structures - Lecture 10

Mihai Herda

12.01.2012

1 Succinct Data Structures

We now look at the problem of representing data structures very space efficiently. A data structure is called *succinct* if its space occupancy is $\log_2 U(n) \cdot (1 + o(1))$ bits, if there are $U(n)$ objects of size n in the universe. Note that this is already the space needed to *distinguish* between the objects in the universe, hence this space is asymptotically *optimal* in the Kolomogorov sense.

Example 1.

1. Permutations *of* $[1, n]$*:* $U(n) = n!$

   $\Rightarrow \lg U(n) = \lg(\sqrt{2\pi n}(\frac{n}{e})^n) = n \lg n - \Theta(n)$

   Hence storing permutations in length-n *arrays is succinct.*

2. *Strings over* $\sum = \{1, \ldots, \sigma\}$ *of length* n*:* $U(n) = \sigma^n$

   $\Rightarrow \lg U(n) = n \lg \sigma$

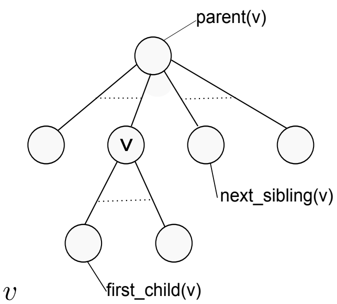
   Hence storing strings as arrays of chars is succinct (assuming all char codes are being used).

3. *Ordered trees of* n *nodes:* $U(n) =$ *with Catalan number* $\approx \frac{4^n}{n^{\frac{3}{2}}\sqrt{\pi}} \Rightarrow \lg U(n) \approx 2n - \Theta(\lg n)$

   Hence storing trees in a pointer-based representation is asymptotically not optimal*, hence* not succint*.*

1.1 Succinct Trees

The aim is to represent a static ordered tree of n nodes using $2n + o(n)$ bits, while still being able to work with the tree as if it were stored in a pointer-based representation.
In particular, we want to support the following operations, all in $O(1)$ time:

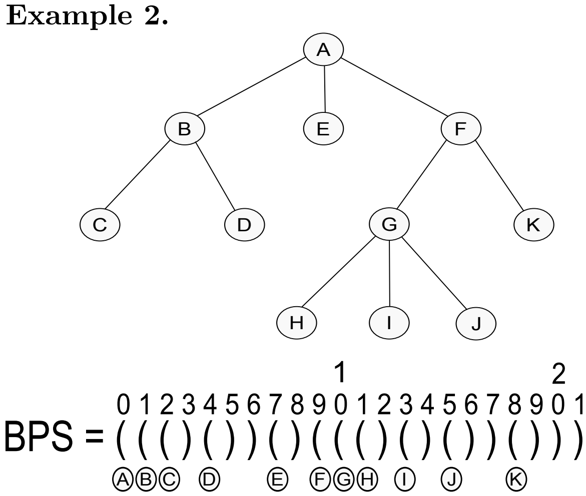


- $parent(v)$: parent node of v
- $first\_child(v)$: leftmost child of v
- $next\_sibling(v)$: next sibling of v
- $is\_leaf(v)$: test if v is a leaf
- $is\_ancestor(u, v)$: test if u is ancestor of v
- $subtree\_size(v)$: number of nodes below v
- $depth(v)$: number of nodes on the root-to-v path

1.1.1 Balanced Parentheses (BP)

We represent the tree T as its *sequence of balanced parentheses*. This is obtained during a DFS through T, writing an opening parenthesis '(' when a node is encountered for the first time, and a closing parenthesis ')' when it is seen last. Then a node can be identified by a pair of matching parentheses '(...)'. We adopt the convention of identifying nodes by the position of the opening parenthesis in the BP sequence.

Example 2.

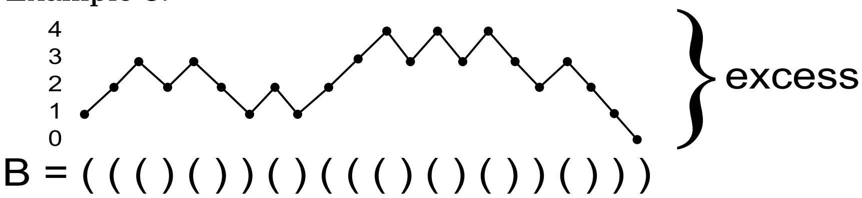


We can represent the BPS in a bit-string B of length $2n$ by encoding '(' as '1' and ')' as '0'. Let us define the *excess* of a position $1 \leq i \leq 2$ in B as the number of ('s minus the number of)'s in the prefix of B up to position i:

Definition 1. $excess(B, i) = |\{j \leq i : B[j] =' ('\}| - |\{j \leq i : B[j] =')'\}|$

Note that the excess is never negative and it is equal to 0 only for the last position $i = 2n$.

Example 3.



1.1.2 Reduction to Core Operations

The navigational operations can be reduced to the following 4 core operations:

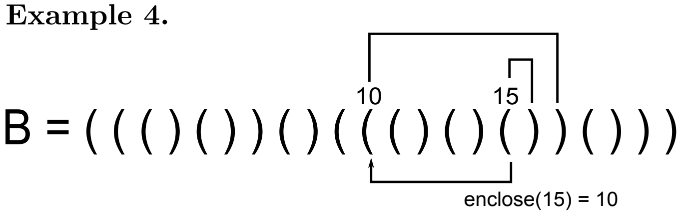
$rank_{(}(B, i)$ = number of ('s in $B[0, i]$

$rank_{)}(B, i)$ = number of)'s in $B[0, i]$

$findclose(B, i)$ = position of matching closing parenthesis if $B[i] =' ('$

$enclose(B, i)$ = position j of the opening parenthesis such that $(j, findclose(j))$ encloses $(i, findclose(i))$ most tightly.

Example 4.



The operations can be expressed as follows:

- $parent(i) = enclose(B, i)$ (if $i \neq 0$, otherwise $root$)
- $first\_child(i) = i + 1$ (if $B[i] =' ('$, otherwise $leaf$)
- $next\_sibling(i) = findclose(i) + 1$ (if $B[findclose(i) + 1] =' ('$, otherwise i is last sibling)
- $is\_leaf(i)$ = true iff $B[i + 1] =')'$
- $is\_ancestor(i, j)$ = true iff $i \leq j \leq findclose(B, i)$
- $subtree\_size(i) = (findclose(i) - i + 1)/2$
- $depth(i) = rank_{(}(B, i) - rank_{)}(B, i)$

Note also $excess(i) = rank_{(}(B, i) - rank_{)}(B, i)$ for *all* positions i (not only for positions of opening parantheses, where $excess(i) = depth(i)$).
In order to jump directly to the opening parenthesis of the i'th node, we define the operation *select* as follows:

Definition 2. $select_{(}(B, i)$ = *position of i'th '(' in B*

1.2 Rank and Select

We start with rank and select, as they will also be used as subroutines for findclose and enclose. Recall that we represent the BPS as a bit-vector, hence we can formulate the following task:

given: a bit-vector B of length n

compute: a data structure that supports $rank_1(B, i)$ and $select_1(B, i)$ for all $i \leq j \leq n$. The size of the data structure should be asymptotically smaller than the size of B.

For rank, we divide B into *blocks* of length $s = \frac{\lg n}{2}$ and *superblocks* of length $s' = s^2$.
In a table $SBlkRank[0, n/s']$, we store the answers to rank for super-blocks, and in $BlkRank[0, n/s]$ the same for blocks, but only relative to the beginning of the super-block:

$$SBlkRank[i] = rank_1(B, i \cdot s' - 1)$$

$$BlkRank[i] = rank_1(B, i \cdot s - 1) - rank_1(B, \left\lfloor \tfrac{i}{s} \right\rfloor s' - 1)$$

Example 5.

```
                     1                   2
    0 1 2|3 4 5|6 7 8|9 0 1|2 3 4|5 6 7|8 9 0|1 2 3|4 5 6
B = 1 1 1|0 1 0|0 1 0|1 1 1|0 1 0|1 0 0|1 0 0|0 0 1|1 0 0
                                        s     s'
```

SBlkRank = 0 5 10

BlkRank = 0 3 4 | 0 3 4 | 0 1 2

Inblock =

pattern	i = 0 1 2
0 0 0	0 0 0
0 0 1	0 0 1
0 1 0	0 1 1
0 1 1	0 1 2
1 0 0	1 1 1
1 0 1	1 1 2
1 1 0	1 2 2
1 1 1	1 2 3

We also store a *lookup table* $Inblock[0, 2^s - 1][0, s - 1]$ where $inblock[pattern][i] = rank_1(pattern, i)$ for all bit patterns of length s and all $0 \leq i \leq s$. Then

$$rank_1(B, i) = SBlkRank[\lfloor \tfrac{i}{s'} \rfloor] + BlkRank[\lfloor \tfrac{i}{s} \rfloor] + Inblock[B[\underbrace{\lfloor \frac{i}{s} \rfloor s}_{\text{start of } i\text{'s block}}, \underbrace{\lceil \frac{i+1}{s} \rceil s - 1}_{\text{end of } i\text{'s block}}]][i - \lfloor \tfrac{i}{s} \rfloor s]$$

The sizes of the data structures are order of

$$|SBlkRank| = \underbrace{\frac{n}{s'}}_{\#\text{superblocks}} \times \underbrace{\lg n}_{\#\text{bits for value}} = \frac{n}{\lg n},$$

$$|BlkRank| = \tfrac{n}{s} \times \lg s' = \tfrac{n \lg \lg n}{\lg n}, \text{ and}$$

$$|Inblock| = 2^s \times s \times \lg s = \sqrt{n} \lg n \lg \lg n,$$

all $o(n)$ bits.

1.3 Recommended Reading

- R.F.Geary, N.Rahman, R. Raman, V.Raman: A Simple Optimal Representation for Balanced Parantheses. Theor. Comp. Sci. 368(3): 231-246, 2006.
- J. I. Munro, V. Raman: Succinct Representation of Balanced Parenthesis and Static Trees. SIAM J. Comput 31(3): 762 - 776, 2001.

There is a vast amount of literature on succinct tree representations, focusing on enhancing the set of supported operations(i-th_child, lca, level-ancestor, ...), dynamization(insert/delete nodes), lowering the redundancy(the $o(n)$-term), etc. A good pointer to recent developments is:

- K. Sadakane, G. Navarro: Fully-Functional Succinct Trees. Proc. SODA: 134-149, 2010.