

Advanced Data Structures

Lecture 02: Succinct Trees

Florian Kurpicz

The slides are licensed under a Creative Commons Attribution-ShareAlike 4.0 International License © ⓘ ⓘ: www.creativecommons.org/licenses/by-sa/4.0 | commit 3c6d2d4 compiled at 2022-05-02-08:36



<https://pingo.scc.kit.edu/578813>

Recap: Rank Queries on Bit Vectors (1/2)

$\text{rank}_\alpha(i)$ # of α s before i

$\text{select}_\alpha(j)$ position of j -th α

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

Recap: Rank Queries on Bit Vectors (1/2)

$\text{rank}_\alpha(i)$ # of α s before i

$\text{select}_\alpha(j)$ position of j -th α

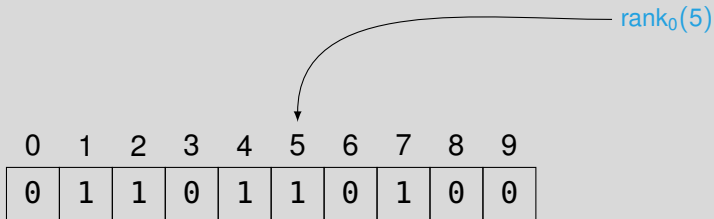
$\text{rank}_0(5)$

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

Recap: Rank Queries on Bit Vectors (1/2)

$\text{rank}_\alpha(i)$ # of α s before i

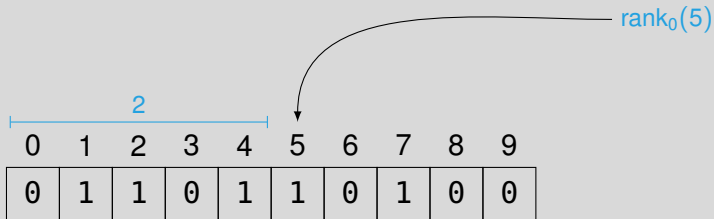
$\text{select}_\alpha(j)$ position of j -th α



Recap: Rank Queries on Bit Vectors (1/2)

$\text{rank}_\alpha(i)$ # of α s before i

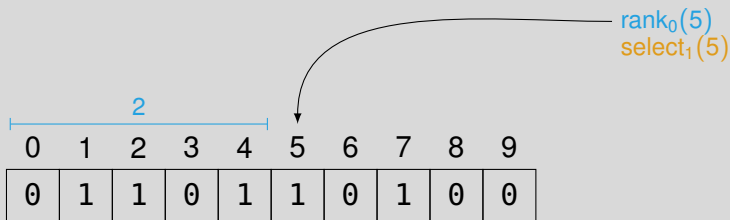
$\text{select}_\alpha(j)$ position of j -th α



Recap: Rank Queries on Bit Vectors (1/2)

$\text{rank}_\alpha(i)$ # of α s before i

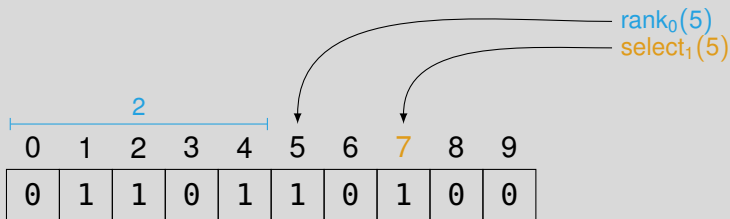
$\text{select}_\alpha(j)$ position of j -th α



Recap: Rank Queries on Bit Vectors (1/2)

$\text{rank}_\alpha(i)$ # of α s before i

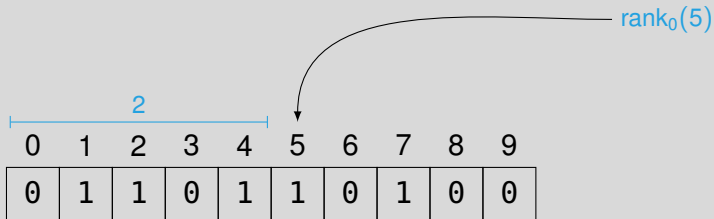
$\text{select}_\alpha(j)$ position of j -th α



Recap: Rank Queries on Bit Vectors (1/2)

$\text{rank}_\alpha(i)$ # of α s before i

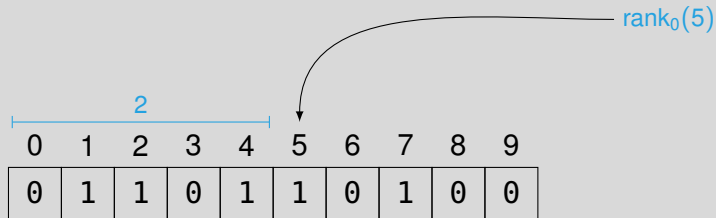
$\text{select}_\alpha(j)$ position of j -th α



Recap: Rank Queries on Bit Vectors (1/2)

$\text{rank}_\alpha(i)$ # of α s before i

$\text{select}_\alpha(j)$ position of j -th α

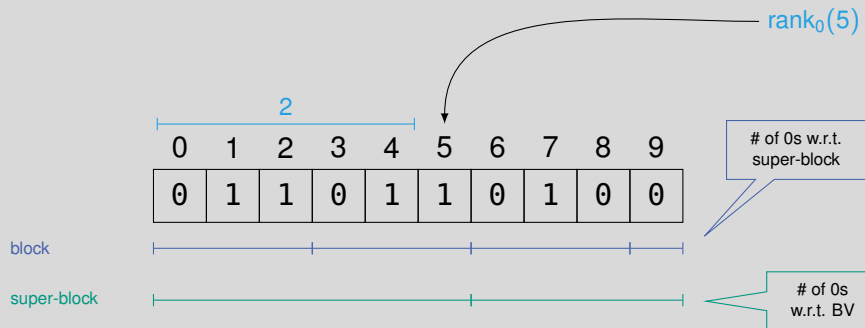


of 0s
w.r.t. BV

Recap: Rank Queries on Bit Vectors (1/2)

$\text{rank}_\alpha(i)$ # of α s before i

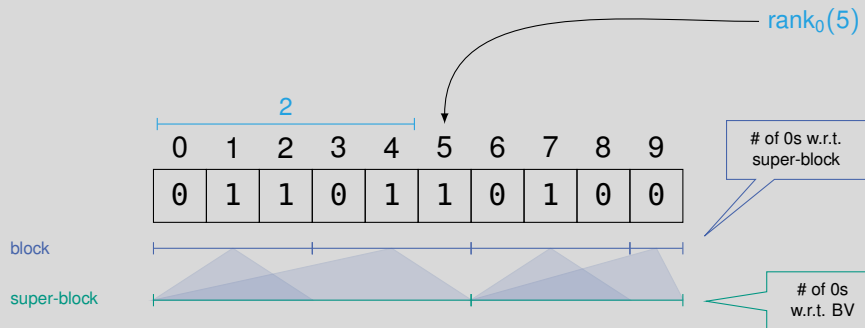
$\text{select}_\alpha(j)$ position of j -th α



Recap: Rank Queries on Bit Vectors (1/2)

$\text{rank}_\alpha(i)$ # of α s before i

$\text{select}_\alpha(j)$ position of j -th α



Recap: Rank Queries on Bit Vectors (2/2)

Lemma: Binary Rank- and Select-Queries

Given a bit vector of size n , there exist data structures that can be computed in time $O(n)$ of size $o(n)$ bits that can answer rank and select queries on the bit vector in $O(1)$ time

Recap: Rank Queries on Bit Vectors (2/2)

Lemma: Binary Rank- and Select-Queries

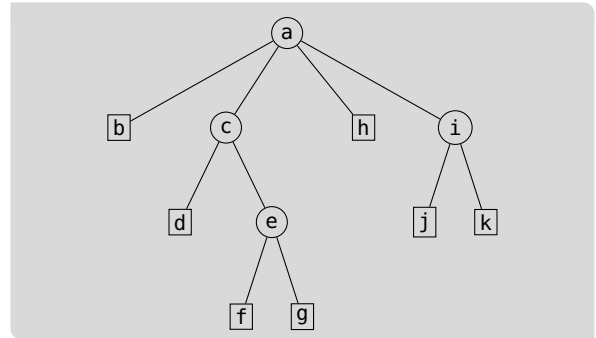
Given a bit vector of size n , there exist data structures that can be computed in time $O(n)$ of size $o(n)$ bits that can answer rank and select queries on the bit vector in $O(1)$ time

Word RAM

- unlimited memory
- words of size w $w = \Theta(\log n)$
- constant time load and store
- constant time bit operations on words

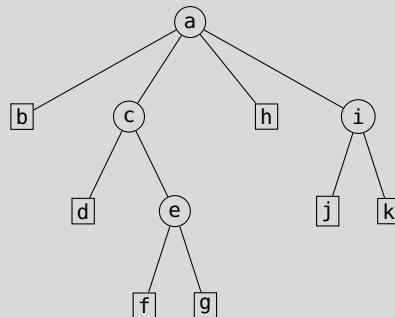
Plan for Today

- represent tree with n nodes using $2n$ bits
- make succinct tree fully-functional using additional $o(n)$ bits



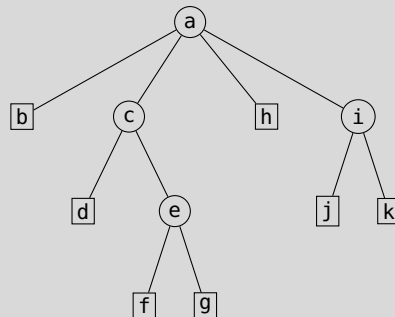
Plan for Today

- represent tree with n nodes using $2n$ bits
 - make succinct tree fully-functional using additional $o(n)$ bits
-
- trees are important
 - searching for keys
 - maintaining directories
 - representations of parsings
 - ...



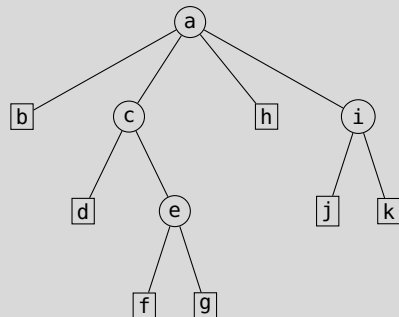
Plan for Today

- represent tree with n nodes using $2n$ bits
 - make succinct tree fully-functional using additional $o(n)$ bits
-
- trees are important
 - searching for keys
 - maintaining directories
 - representations of parsings
 - ...
-
- different representations
 - supporting different operations



Plan for Today

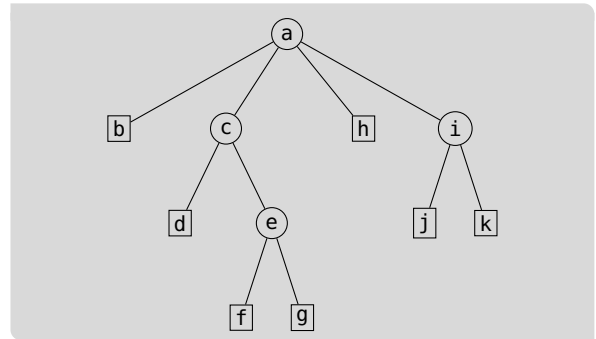
- represent tree with n nodes using $2n$ bits
 - make succinct tree fully-functional using additional $o(n)$ bits
-
- trees are important
 - searching for keys
 - maintaining directories
 - representations of parsings
 - ...
-
- different representations
 - supporting different operations



Handout

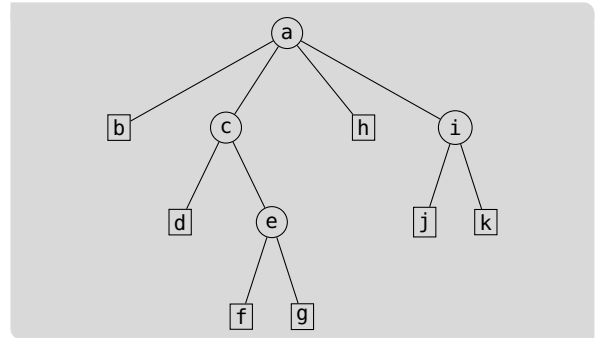
Preliminaries

- a tree is an acyclic connected graph $G = (V, E)$ with a root $r \in V$
- degree δ is the number of children
- leaves have degree 0
- depth of a node is the length of the path from the root to that node



Level Ordered Unary Degree Sequence (1/2) [Jac88]

- represent tree level-wise
- use ≤ 2 bits per node



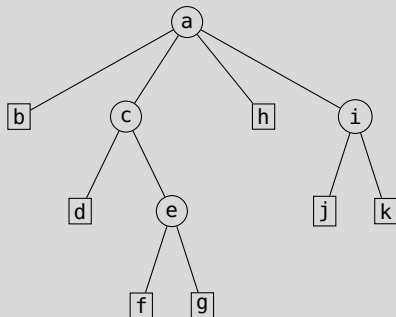
Level Ordered Unary Degree Sequence (1/2) [Jac88]

- represent tree level-wise
- use ≤ 2 bits per node

Definition: LOUDS

Starting at the root, all nodes on the **same depth**

- are visited from left to right and
- for node v , $\delta(v)$ 1's followed by a 0 are appended to the bit vector that contains an initial 10



Level Ordered Unary Degree Sequence (1/2) [Jac88]

- represent tree level-wise
- use ≤ 2 bits per node

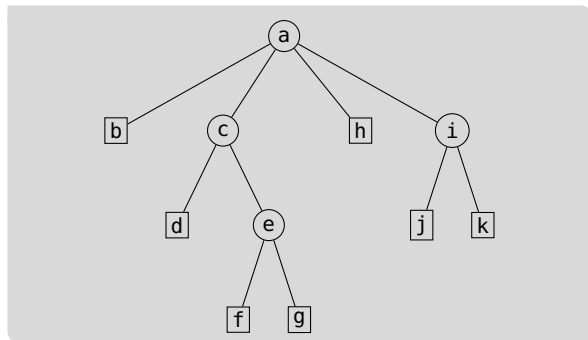
Definition: LOUDS

Starting at the root, all nodes on the **same depth**

- are visited from left to right and
- for node v , $\delta(v)$ 1's followed by a 0 are appended to the bit vector that contains an initial 10

Lemma: Space Usage of LOUDS

Representing a tree with n nodes requires $2n + 1$ bits using LOUDS



Level Ordered Unary Degree Sequence (1/2) [Jac88]

- represent tree level-wise
- use ≤ 2 bits per node

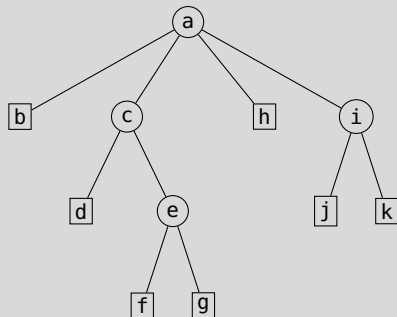
Definition: LOUDS

Starting at the root, all nodes on the **same depth**

- are visited from left to right and
- for node v , $\delta(v)$ 1's followed by a 0 are appended to the bit vector that contains an initial 10

Lemma: Space Usage of LOUDS

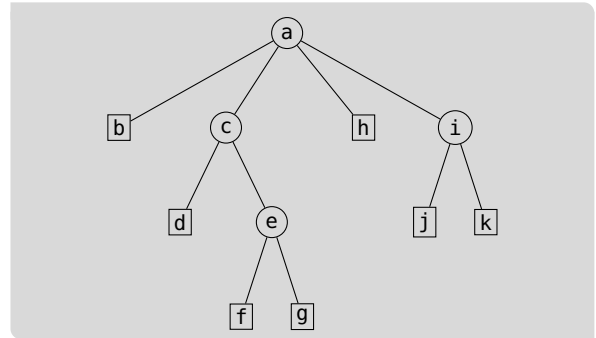
Representing a tree with n nodes requires $2n + 1$ bits using LOUDS



- write down the LOUDS representation of this example tree

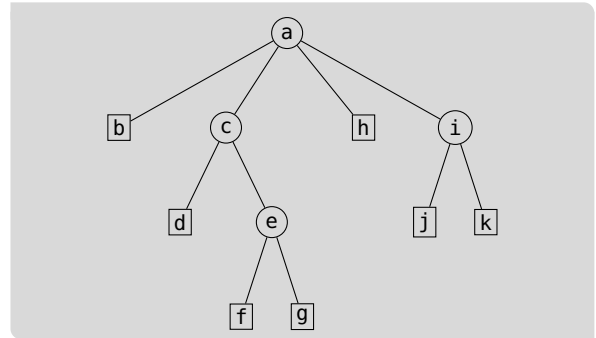
Level Ordered Unary Degree Sequence (2/2)

1011110



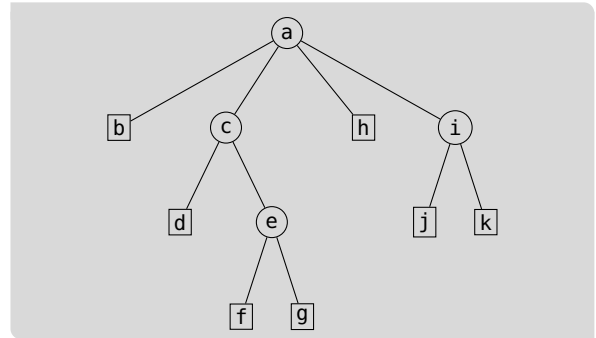
Level Ordered Unary Degree Sequence (2/2)

101111001100110



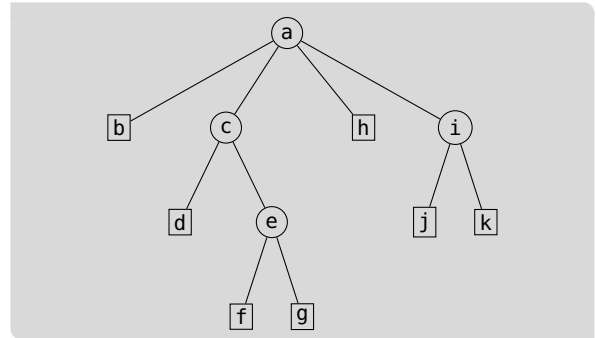
Level Ordered Unary Degree Sequence (2/2)

101111001100110011000



Level Ordered Unary Degree Sequence (2/2)

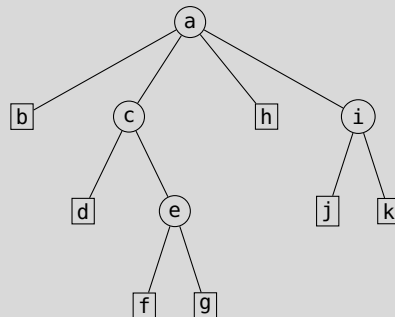
101111100110011001100000



Level Ordered Unary Degree Sequence (2/2)

ab ch id ejkfg
 10111100110011001100000

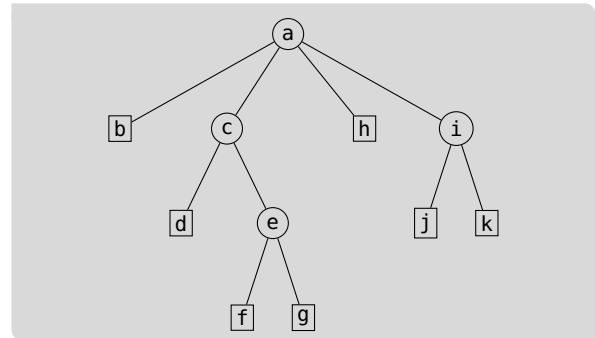
- node start at pertinent 0



What is Fully-Functional?

Operations

- degree i is leaf
- i -th child
- parent
- subtree size

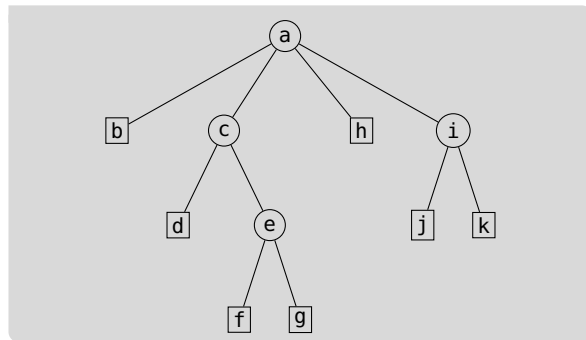


What is Fully-Functional?

Operations

- degree i is leaf
- i -th child
- parent
- subtree size

- depth
- lowest common ancestor
- rank (pre- or post-order)
- ...




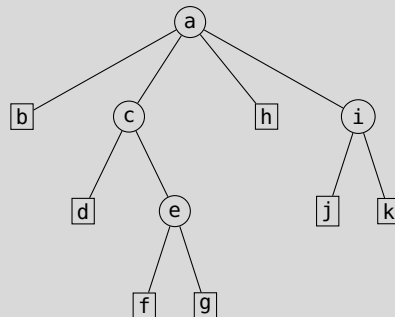
Making LOUDS Fully-Functional

```

ab ch id ejkfg
10111100110011001100000
  
```

- degree of p : $p - \text{select}_0(\text{rank}_0(p)) - 1$

- explanation on the board 




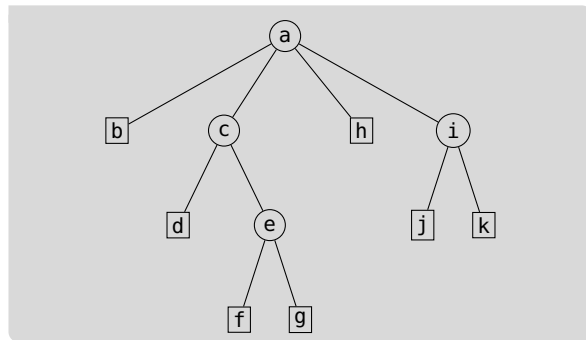
Making LOUDS Fully-Functional

```

ab ch id ejkfg
10111100110011001100000
  
```

- degree of p : $p - \text{select}_0(\text{rank}_0(p)) - 1$
- i -th child of p :
 $\text{select}_0(\text{rank}_1(\text{select}_0(\text{rank}_0(p))) + i + 1)$

- explanation on the board 




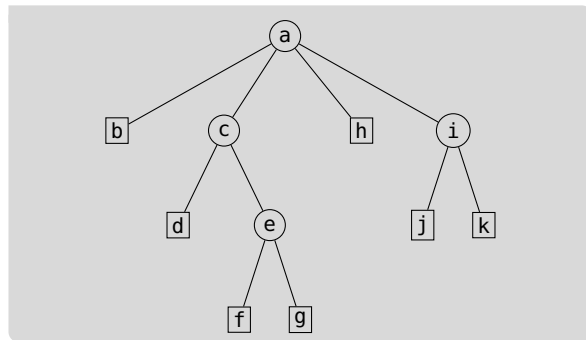
Making LOUDS Fully-Functional

```

ab ch id ejkfg
10111100110011001100000
  
```

- degree of p : $p - \text{select}_0(\text{rank}_0(p)) - 1$
- i -th child of p :
 $\text{select}_0(\text{rank}_1(\text{select}_0(\text{rank}_0(p))) + i + 1)$
- parent of p :
 $\text{select}_0(\text{rank}_0(\text{select}_1(\text{rank}_0(p))) + 1)$

- explanation on the board 





Making LOUDS Fully-Functional

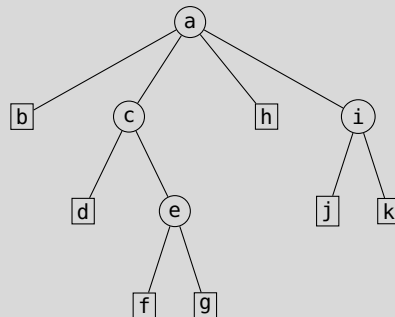
```

ab ch id ejkfg
10111100110011001100000
  
```

- degree of p : $p - \text{select}_0(\text{rank}_0(p)) - 1$
- i -th child of p :
 $\text{select}_0(\text{rank}_1(\text{select}_0(\text{rank}_0(p))) + i + 1)$
- parent of p :
 $\text{select}_0(\text{rank}_0(\text{select}_1(\text{rank}_0(p))) + 1)$

- explanation on the board 

- subtree size  **PINGO**



From Bit Vectors to Parentheses


- instead of 0 and 1
 - use (and)
-
- requires the same space
 - can add relation between parentheses

From Bit Vectors to Parentheses

- instead of 0 and 1
- use (and)

- requires the same space
- can add relation between parentheses

Definition: Balanced String of Parentheses


A string of parentheses is balanced, if for each left parenthesis there exist unique right parenthesis to its right 

From Bit Vectors to Parentheses

- instead of 0 and 1
- use (and)

- requires the same space
- can add relation between parentheses

Definition: Balanced String of Parentheses

A string of parentheses is balanced, if for each left parenthesis there exist unique right parenthesis to its right 


- *findclose(i)*: find the right parenthesis matching the left parenthesis at position i

From Bit Vectors to Parentheses

- instead of 0 and 1
- use (and)

- requires the same space
- can add relation between parentheses

Definition: Balanced String of Parentheses

A string of parentheses is balanced, if for each left parenthesis there exist unique right parenthesis to its right 


- *findclose*(i): find the right parenthesis matching the left parenthesis at position i
- *findopen*(i): find the left parenthesis matching the right parenthesis at position i

From Bit Vectors to Parentheses

- instead of 0 and 1
- use (and)

- requires the same space
- can add relation between parentheses

Definition: Balanced String of Parentheses

A string of parentheses is balanced, if for each left parenthesis there exist unique right parenthesis to its right 


- *findclose*(i): find the right parenthesis matching the left parenthesis at position i
- *findopen*(i): find the left parenthesis matching the right parenthesis at position i
- *excess*(i): find the difference between the number of left and right parentheses before position i

From Bit Vectors to Parentheses

- instead of 0 and 1
- use (and)

- requires the same space
- can add relation between parentheses

Definition: Balanced String of Parentheses

A string of parentheses is balanced, if for each left parenthesis there exist unique right parenthesis to its right 


- *findclose*(i): find the right parenthesis matching the left parenthesis at position i
- *findopen*(i): find the left parenthesis matching the right parenthesis at position i
- *excess*(i): find the difference between the number of left and right parentheses before position i
- *enclose*(i): given a parentheses pair with the left parenthesis at position i , return the position of the closest left parenthesis belonging to the parentheses pair enclosing it

From Bit Vectors to Parentheses


- instead of 0 and 1
- use (and)

- requires the same space
- can add relation between parentheses

Definition: Balanced String of Parentheses

A string of parentheses is balanced, if for each left parenthesis there exist unique right parenthesis to its right 

- *findclose*(i): find the right parenthesis matching the left parenthesis at position i
- *findopen*(i): find the left parenthesis matching the right parenthesis at position i
- *excess*(i): find the difference between the number of left and right parentheses before position i
- *enclose*(i): given a parentheses pair with the left parenthesis at position i , return the position of the closest left parenthesis belonging to the parentheses pair enclosing it

- how can we answer *excess* queries  **PINGO**

From Bit Vectors to Parentheses

- all parentheses operations can be answered in $O(1)$ time using $o(n)$ bits space
- here, a little bit simpler

From Bit Vectors to Parentheses

- all parentheses operations can be answered in $O(1)$ time using $o(n)$ bits space
- here, a little bit simpler

- $excess(i) = rank_{“(} (i) - rank_{“)”} (i)$
- $fwd_search(i, d) = \min\{j > i : excess(j) - excess(i - 1) = d\}$
- $bwd_search(i, d) = \max\{j < i : excess(i) - excess(j - 1) = d\}$

From Bit Vectors to Parentheses

- all parentheses operations can be answered in $O(1)$ time using $o(n)$ bits space
- here, a little bit simpler

- $excess(i) = rank_{“(”}(i) - rank_{“)”}(i)$
- $fwd_search(i, d) = \min\{j > i : excess(j) - excess(i - 1) = d\}$
- $bwd_search(i, d) = \max\{j < i : excess(i) - excess(j - 1) = d\}$

- $findclose(i) = fwd_search(i, 0)$
- $findopen(i) = bwd_search(i, 0)$
- $enclose(i) = bwd_search(i, 2)$

From Bit Vectors to Parentheses

- all parentheses operations can be answered in $O(1)$ time using $o(n)$ bits space
- here, a little bit simpler

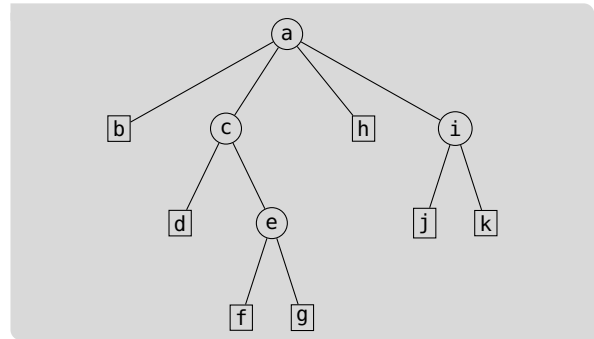
- $excess(i) = rank_{“(”}(i) - rank_{“)”}(i)$
- $fwd_search(i, d) = \min\{j > i : excess(j) - excess(i - 1) = d\}$
- $bwd_search(i, d) = \max\{j < i : excess(i) - excess(j - 1) = d\}$

- $findclose(i) = fwd_search(i, 0)$
- $findopen(i) = bwd_search(i, 0)$
- $enclose(i) = bwd_search(i, 2)$

- can be answered with a [min-max-tree](#) ⓘ later in this course

Balanced Parentheses (1/2) [MR01]

- represent tree as depth-first traversal
- using balanced parentheses



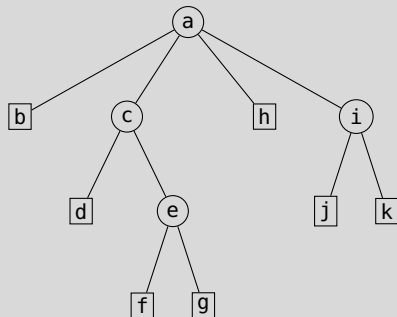
Balanced Parentheses (1/2) [MR01]

- represent tree as depth-first traversal
- using balanced parentheses

Definition: BP

Starting at the root, traverse the tree in **depth-first** order and append a

- left parenthesis if a node is visited the first time
- right parenthesis if a node is visited the last time to the bit vector



Balanced Parentheses (1/2) [MR01]

- represent tree as depth-first traversal
- using balanced parentheses

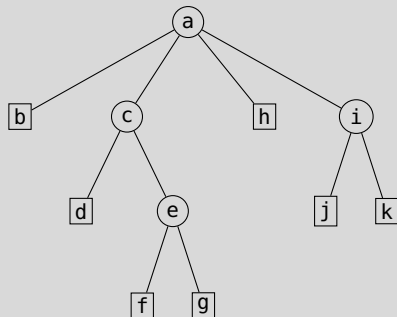
Definition: BP

Starting at the root, traverse the tree in **depth-first** order and append a

- left parenthesis if a node is visited the first time
- right parenthesis if a node is visited the last time to the bit vector

Lemma: Space Usage of BP

Representing a tree with n nodes requires $2n$ bits using *BP*



Balanced Parentheses (1/2) [MR01]

- represent tree as depth-first traversal
- using balanced parentheses

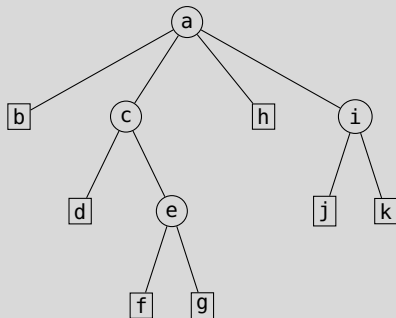
Definition: BP

Starting at the root, traverse the tree in **depth-first** order and append a

- left parenthesis if a node is visited the first time
- right parenthesis if a node is visited the last time to the bit vector

Lemma: Space Usage of BP

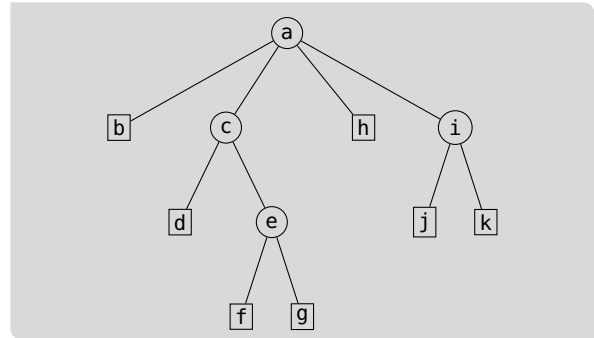
Representing a tree with n nodes requires $2n$ bits using *BP*



- write down the BP representation of this example tree

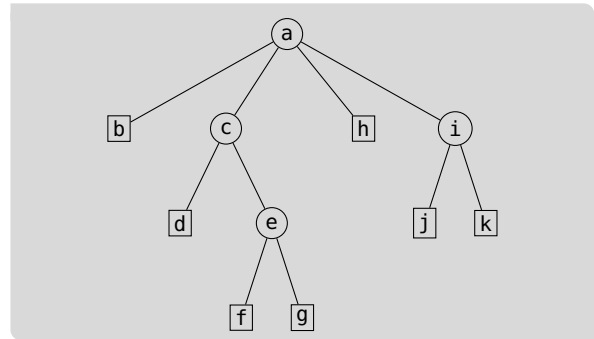
Balanced Parentheses (2/2)

a
(



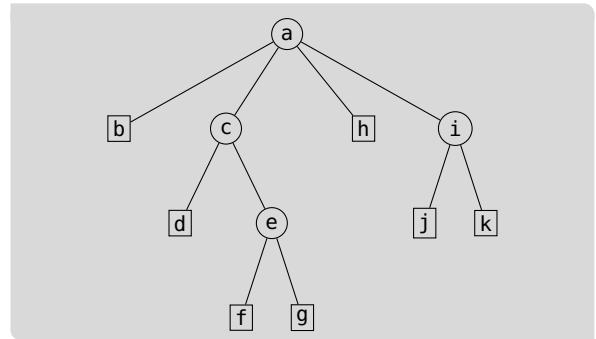
Balanced Parentheses (2/2)

ab
(())



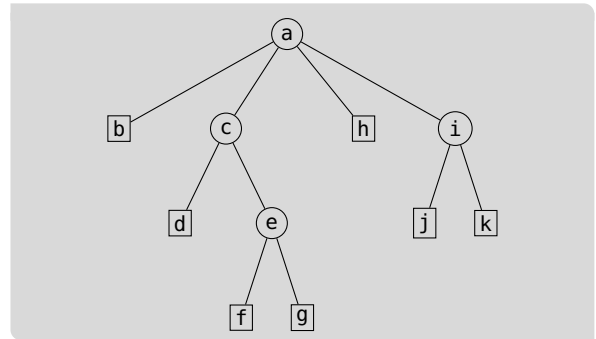
Balanced Parentheses (2/2)

ab cd
(()())



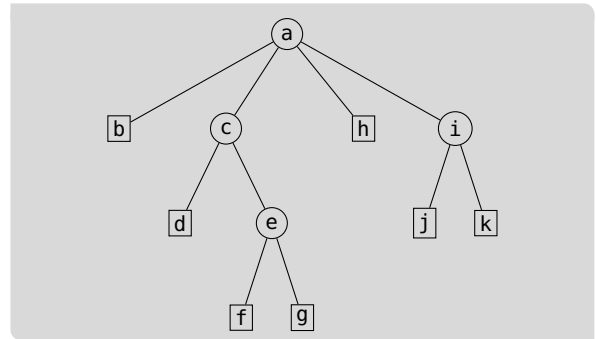
Balanced Parentheses (2/2)

ab cd ef g
((()((()())))



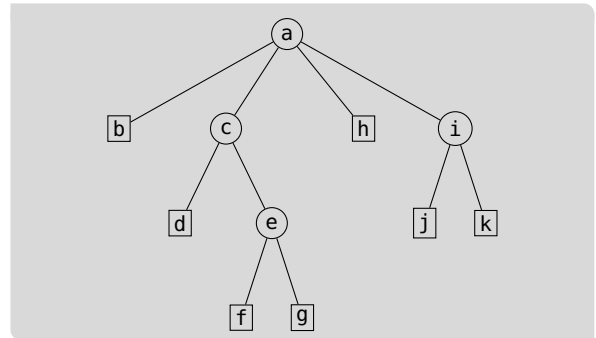
Balanced Parentheses (2/2)

```
ab cd ef g h  
((()((()()))))()
```




Balanced Parentheses (2/2)

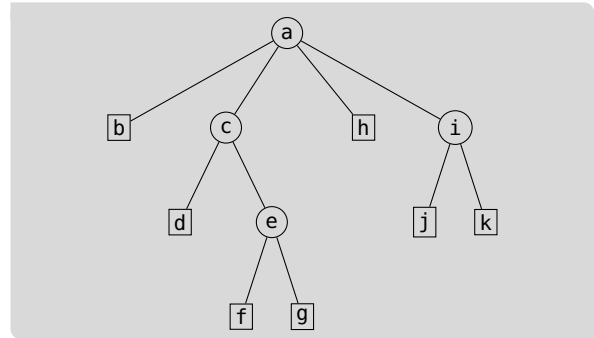
```
ab cd ef g  h ij k  
((()((()())))(()()))
```



Balanced Parentheses (2/2)

```
ab cd ef g  h ij k
((()()()())()((())))
```


- node starts at first parenthesis
- subtree structure is encoded in parentheses 

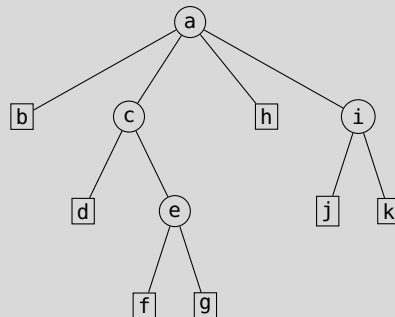


Making BP Fully-Functional

ab cd ef g h ij k
 (()((()((()())))())((()())))


- subtree size of p : $(\text{findclose}(p) - p + 1)/2$

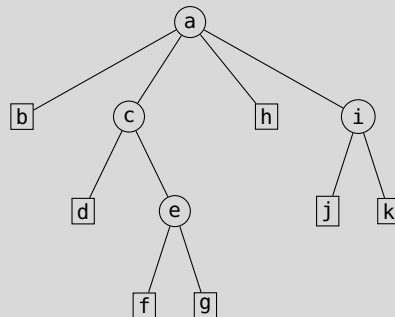
- explanation on the board 



Making BP Fully-Functional

ab cd ef g h ij k
 ((()((()())))(()((()))))


- subtree size of p : $(findclose(p) - p + 1)/2$
- parent of p : $enclose(p)$
- explanation on the board 




Making BP Fully-Functional

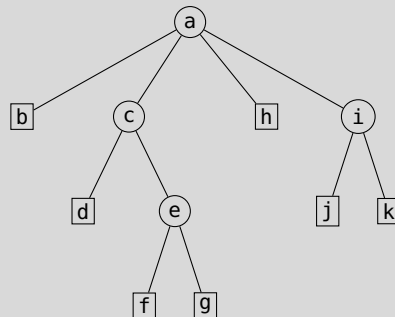
ab cd ef g h ij k
 ((()((()())))(()((()))))

- subtree size of p : $(\text{findclose}(p) - p + 1)/2$
- parent of p : $\text{enclose}(p)$

- explanation on the board 

Complicated Constant Time [NS14]

- degree 
- i -th child



Advantages and Disadvantages of Both Approaches

- LOUDS cannot answer subtree size
 - BP cannot easily answer i -th child and degree
-
- all other operations can be done easily

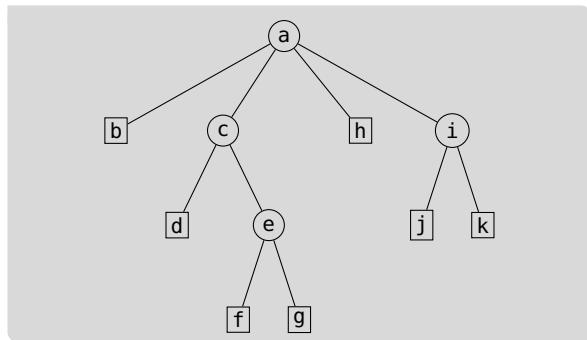
Depth First Unary Degree Sequence (1/2) [Ben+05]

Definition: DFUDS

Starting at the root, traverse tree in **depth-first** order and append

- for node v , $\delta(v)$ left parentheses and
- a right parenthesis if v is visited the first time

to the bit vector that initially contains a left parenthesis **(** to make them balanced



Depth First Unary Degree Sequence (1/2) [Ben+05]

Definition: DFUDS

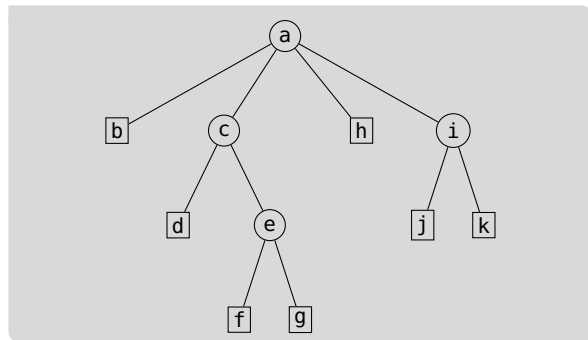
Starting at the root, traverse tree in **depth-first** order and append

- for node v , $\delta(v)$ left parentheses and
- a right parenthesis if v is visited the first time

to the bit vector that initially contains a left parenthesis **(** to make them balanced

Lemma: Space Usage of DFUDS

Representing a tree with n nodes requires $2n$ bits using DFUDS



Depth First Unary Degree Sequence (1/2) [Ben+05]

Definition: DFUDS

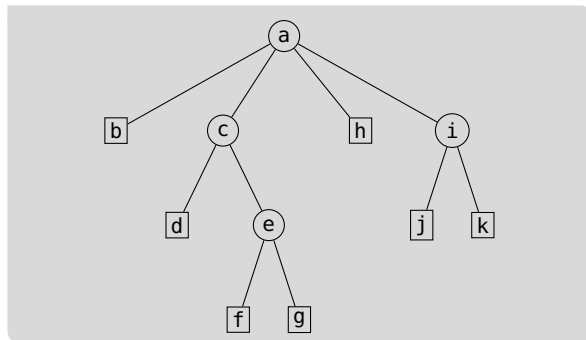
Starting at the root, traverse tree in **depth-first** order and append

- for node v , $\delta(v)$ left parentheses and
- a right parenthesis if v is visited the first time

to the bit vector that initially contains a left parenthesis **(** to make them balanced

Lemma: Space Usage of DFUDS

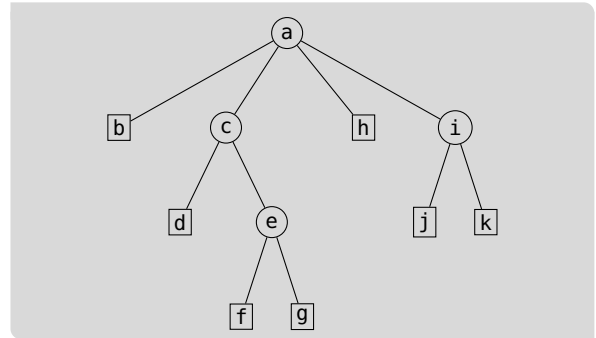
Representing a tree with n nodes requires $2n$ bits using DFUDS



- write down the DFUDS representation of this example tree

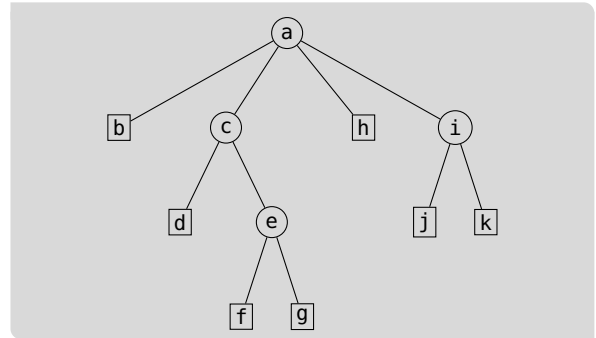
Depth First Unary Degree Sequence (2/2)

a
((((



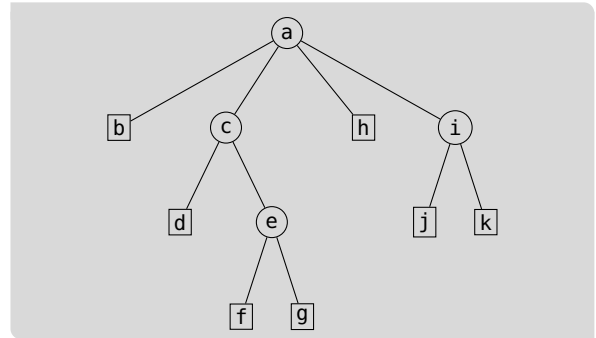
Depth First Unary Degree Sequence (2/2)

a b
 ((((()))



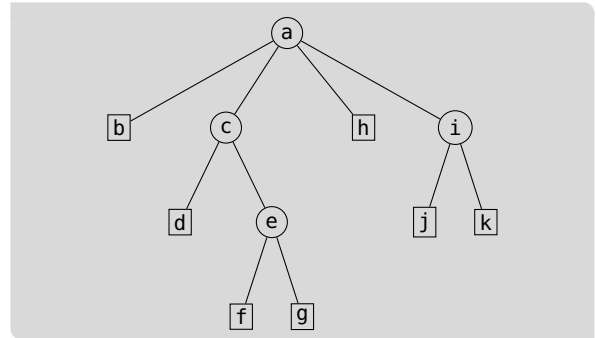
Depth First Unary Degree Sequence (2/2)

a bc
 (((((())) ()))



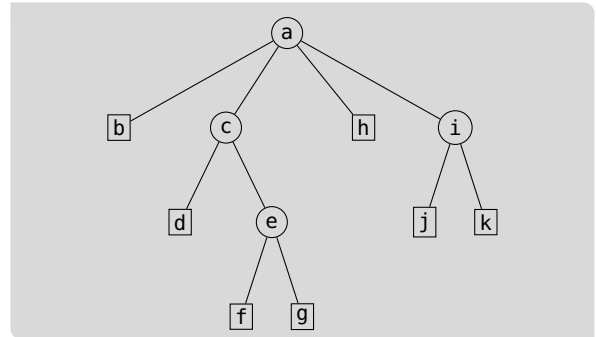
Depth First Unary Degree Sequence (2/2)

a bc d
 ((((())) ())



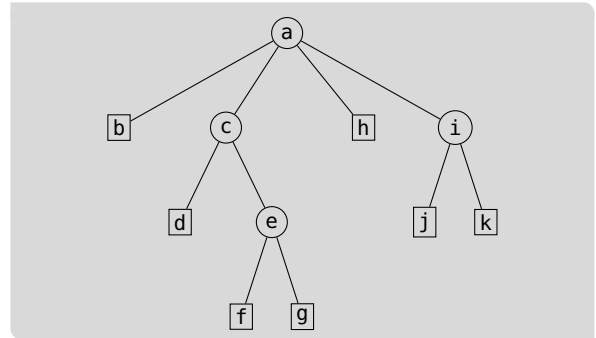
Depth First Unary Degree Sequence (2/2)

a bc de fg
 ((((())())()))



Depth First Unary Degree Sequence (2/2)

a bc de fgh
 (((((()))((()))())))



Conclusion and Outlook

This Lecture

- three succinct tree representations
- different advantages and disadvantages

Advanced Data Structures

BV

succ. trees

Conclusion and Outlook

This Lecture

- three succinct tree representations
- different advantages and disadvantages

- outlook to min-max-trees

Advanced Data Structures

BV

succ. trees

Conclusion and Outlook

This Lecture

- three succinct tree representations
- different advantages and disadvantages

- outlook to min-max-trees

Next Lecture

- dynamic bit vectors and succinct trees
- maybe succinct graphs

Advanced Data Structures

BV

succ. trees

Bibliography I

- [Ben+05] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. “Representing Trees of Higher Degree”. In: *Algorithmica* 43.4 (2005), pages 275–292. DOI: [10.1007/s00453-004-1146-6](https://doi.org/10.1007/s00453-004-1146-6).
- [Jac88] Guy Joseph Jacobson. “Succinct Static Data Structures”. PhD thesis. Carnegie Mellon University, 1988.
- [MR01] J. Ian Munro and Venkatesh Raman. “Succinct Representation of Balanced Parentheses and Static Trees”. In: *SIAM J. Comput.* 31.3 (2001), pages 762–776. DOI: [10.1137/S0097539799364092](https://doi.org/10.1137/S0097539799364092).
- [NS14] Gonzalo Navarro and Kunihiro Sadakane. “Fully Functional Static and Dynamic Succinct Trees”. In: *ACM Trans. Algorithms* 10.3 (2014), 16:1–16:39. DOI: [10.1145/2601073](https://doi.org/10.1145/2601073).