Cache Oblivious Search Trees via Binary Trees of Small Height

Gerth Stolting Brodal
Rolf Fagerberg
Riko Jacob

Lecture Outline

- Motivation
- General idea & Working methods
- van Emde Boas memory layout
- The algorithm
  - Dictionary operations
  - Insertion complexity proof
- Experiments & Results
- Summary

Motivation

- Modern computers contain a hierarchy of memory levels.
- Access time is about 1 [cyc] for registers/ L1 cache, and 100,000 [cyc] for the disk.
- The cost of a memory access depends highly on what is the lowest memory level containing the data.
- The evolution of CPU speed and memory access time indicates that these differences are likely to increase in the future.

Motivation

- Our goal:
  - To find an implementation for binary search tree that tries to minimize cache misses.
  - That algorithm will be cache oblivious.
- By optimizing an algorithm to one unknown memory level, it is optimized to each memory level automatically !

General idea & Working methods

- Definitions:
- A tree T1 can be embedded in another tree T2, if T1 can be obtained from T2 by pruning subtrees.
- Implicit layout - the navigation between a node and its children is done based on address arithmetic, and not on pointers.

General idea & Working methods

- Assume we have a binary search tree.
- Embed this tree in a static complete tree.
- Save this (complete) tree in the memory in a cache oblivious fashion.
- Whenever n is doubled we create a new static tree.

General idea & Working methods

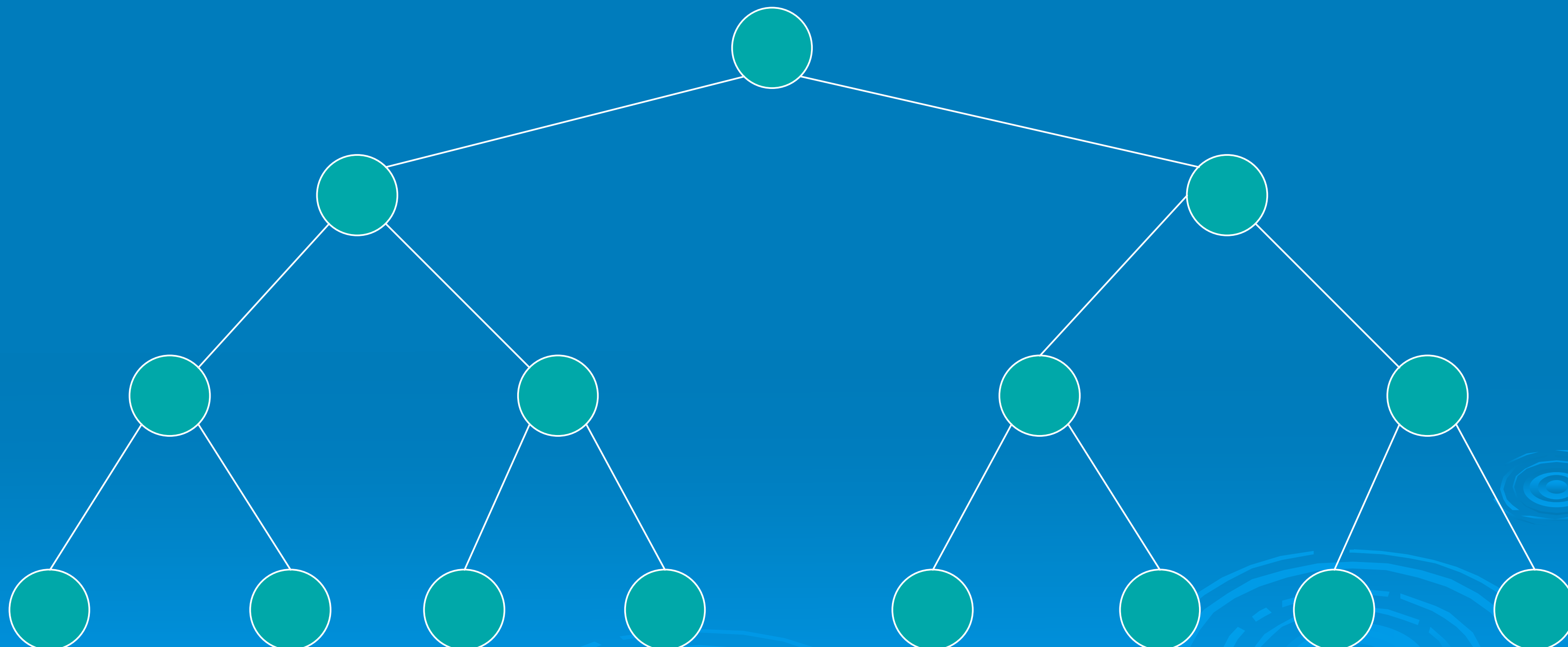
- Advantages:
  - Minimizing memory transfers.
  - Cache obliviousness
  - No pointers – better space utilization:
    - A larger fraction of the structure can reside in lower levels of the memory.
    - More elements can fit in a cache line.
- Disadvantages:
  - Implicit layout: higher instruction count per navigation – slower.

van Emde Boas memory layout

- Recursive definition:
- A tree with only one node is a single node record.
- If a tree T has two or more nodes:
  - Divide T to a top tree T_0 with height [h(T)/2] and a collection of bottom trees $T_1,\ldots,T_k$ with height [h(T)/2] , numbered from left to right.
  - The van Emde Boas layout of T consist of the v.E.B. layout of T_0 followed by the v.E.B. layout of $T_1,\ldots,T_k$

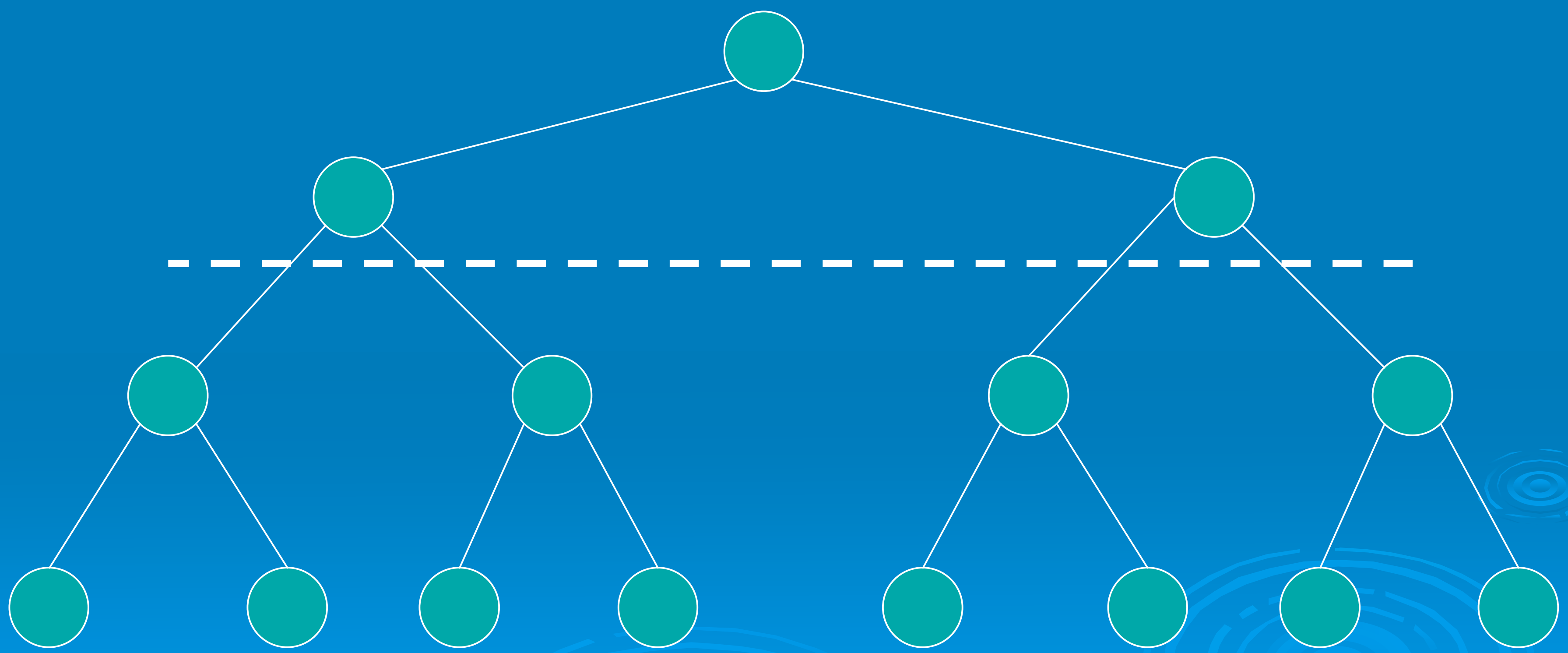
van Emde Boas memory layout

- Example :



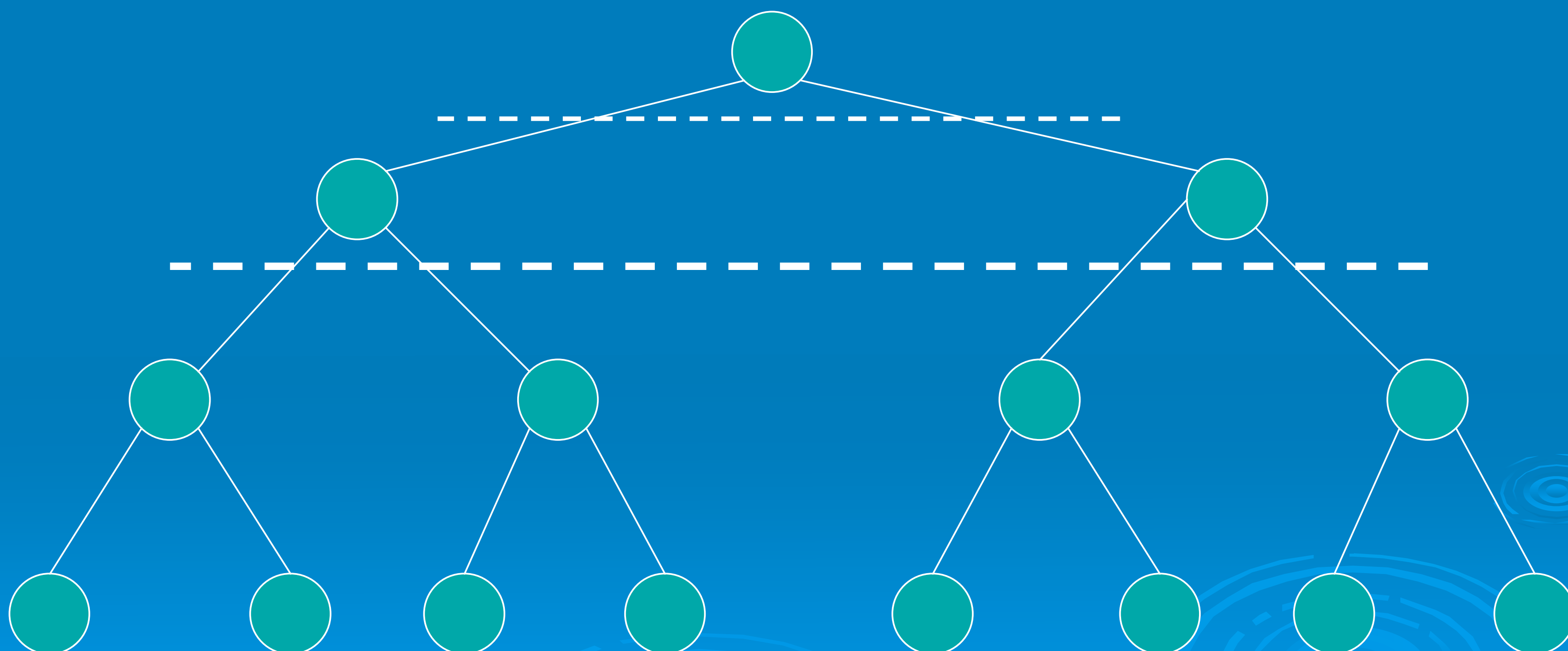
van Emde Boas memory layout

- Example :



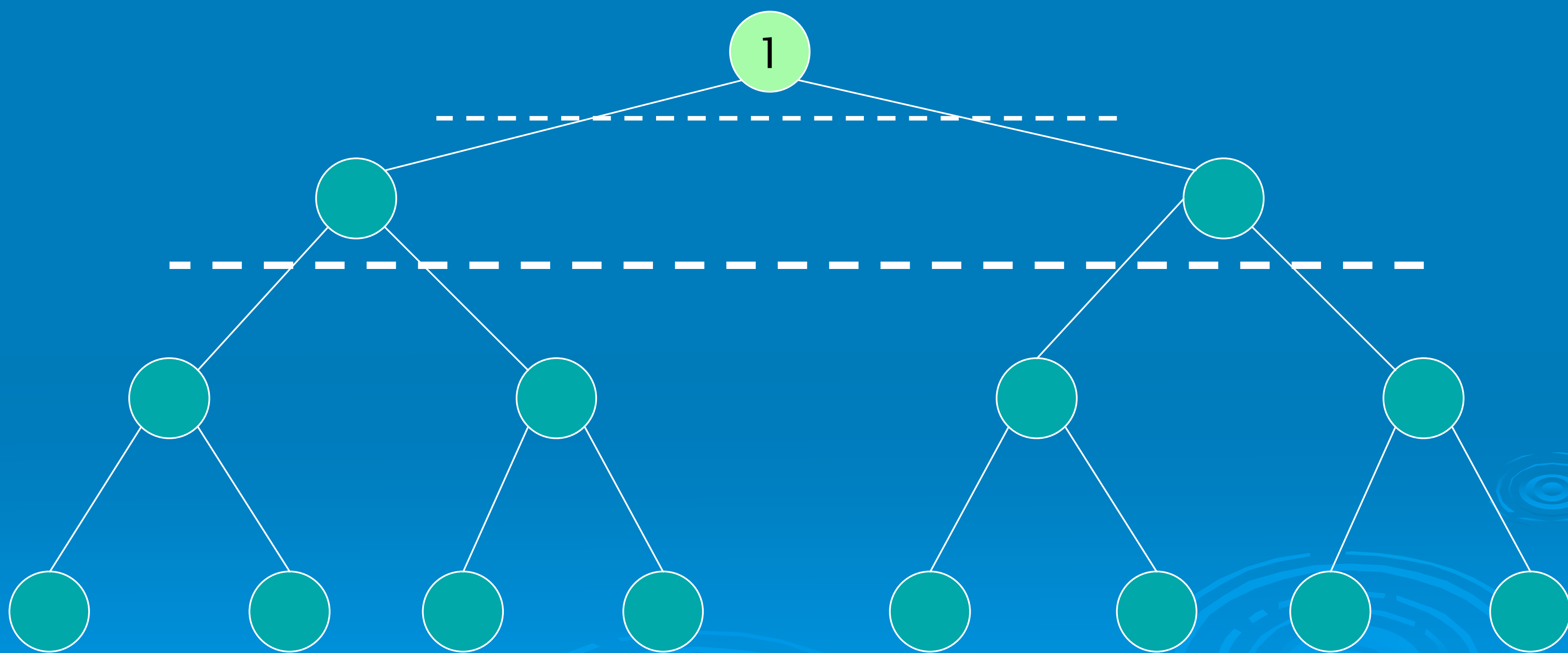
van Emde Boas memory layout

- Example :



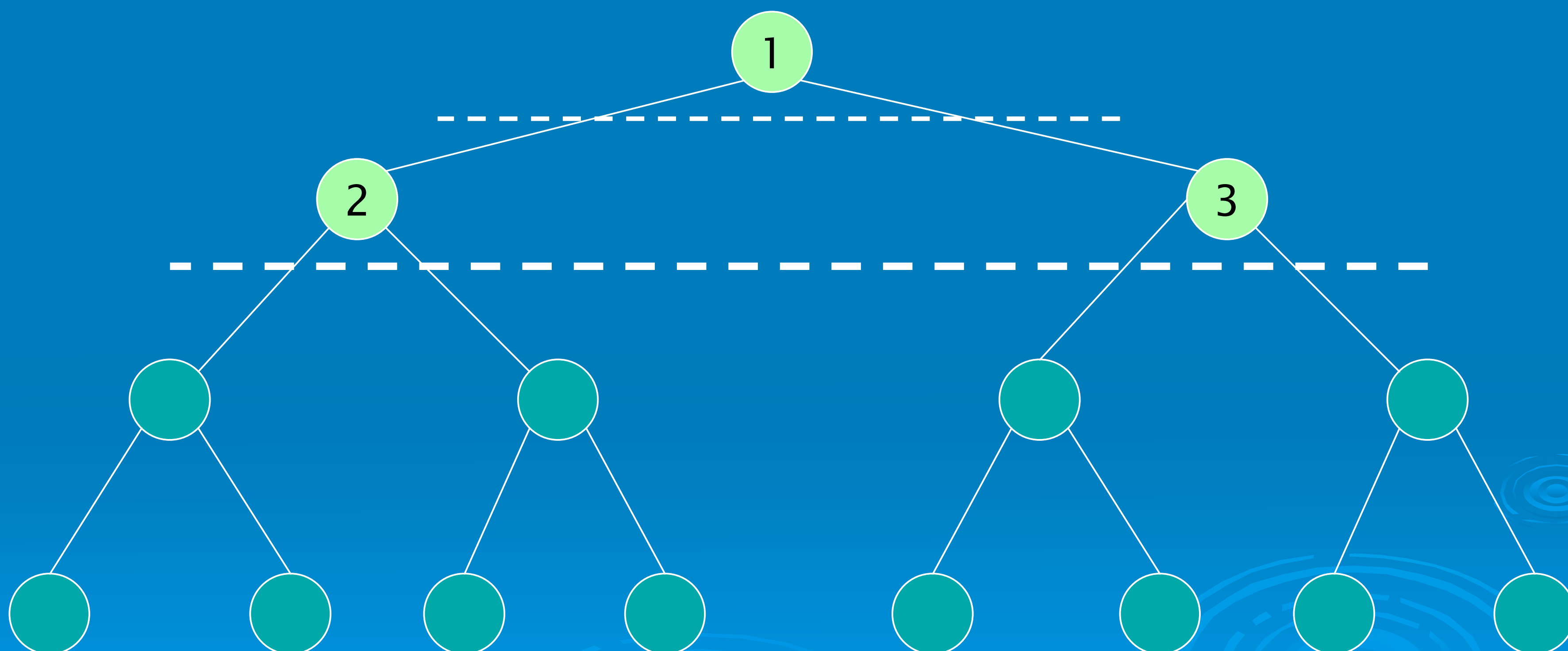
van Emde Boas memory layout

- Example :



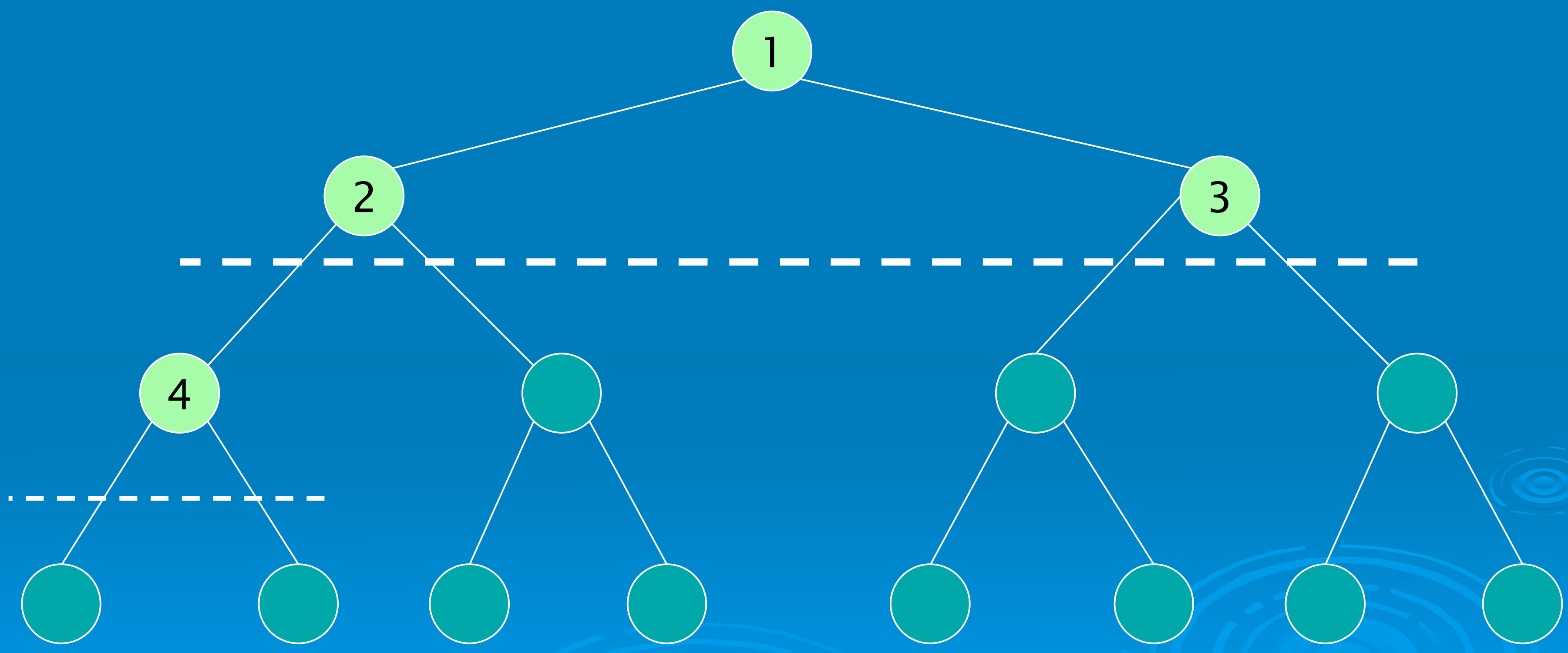
van Emde Boas memory layout

- Example :



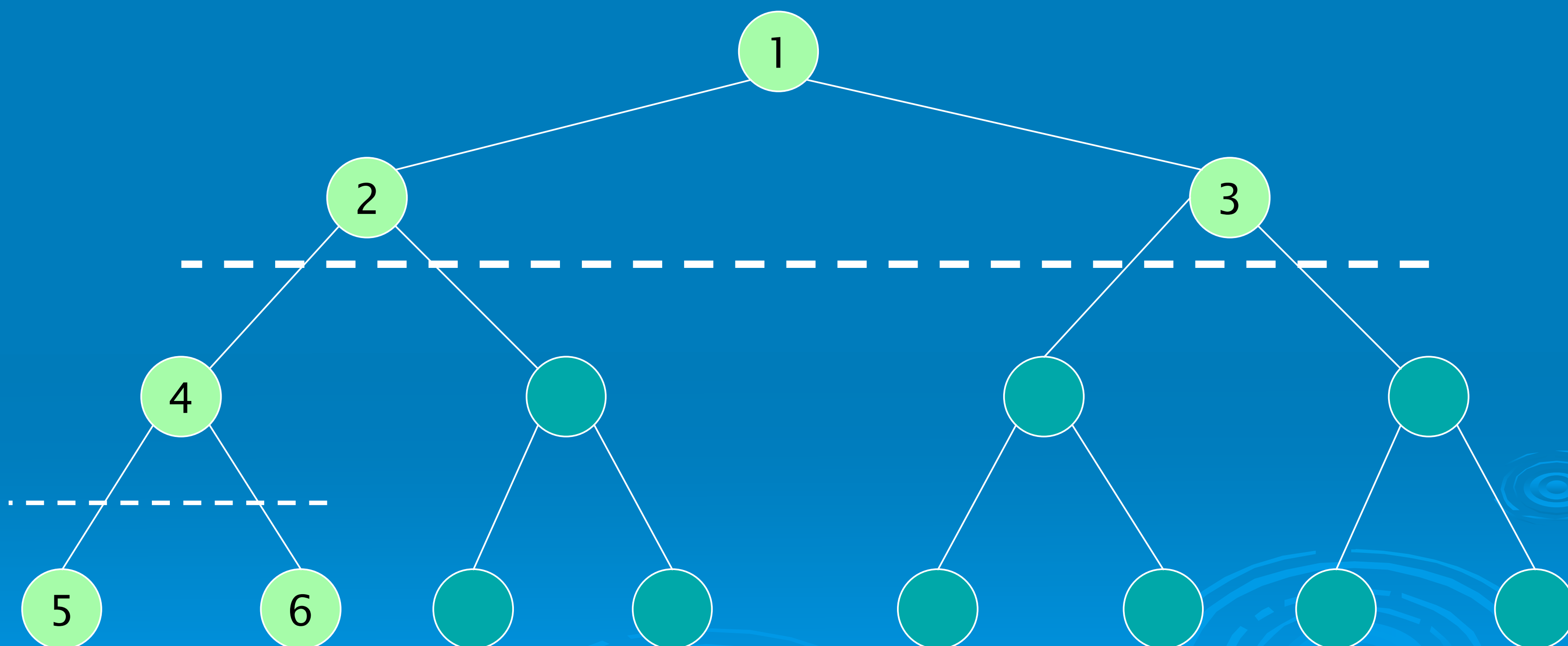
van Emde Boas memory layout

- Example :



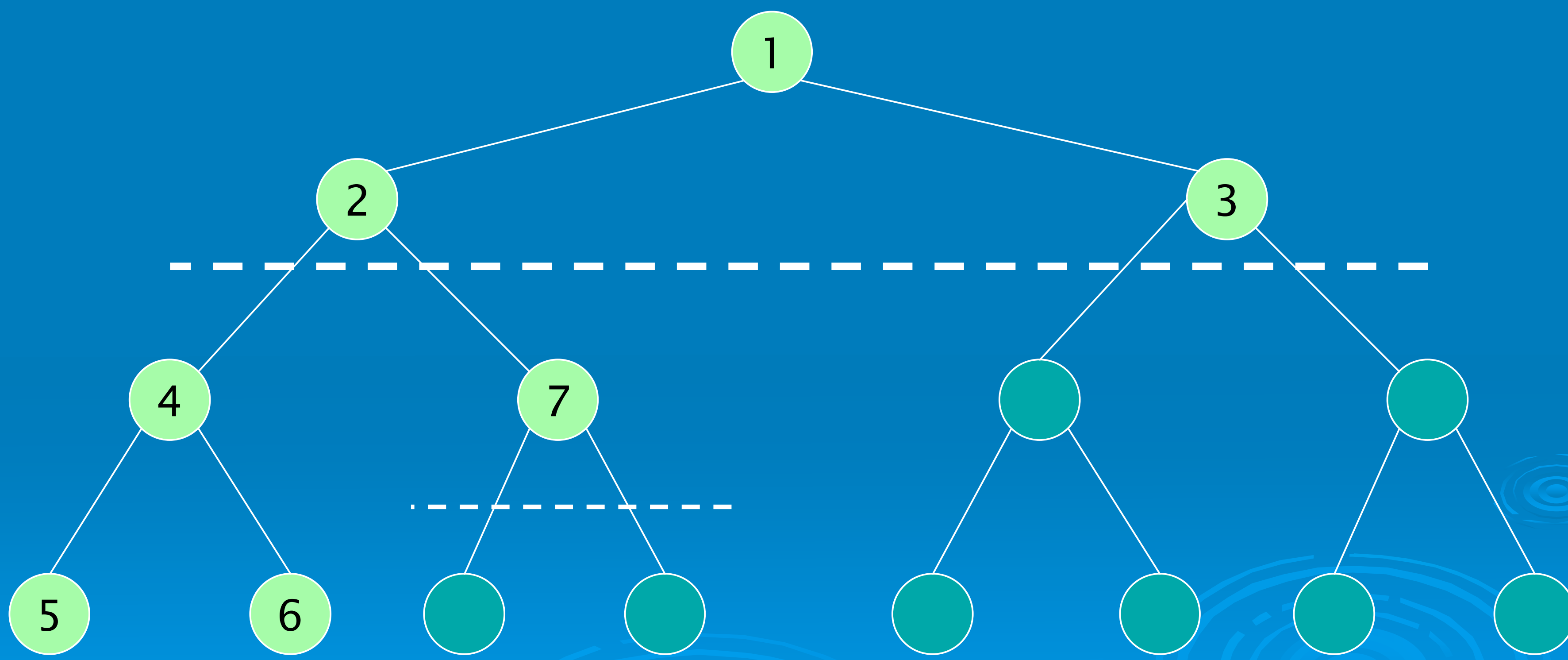
van Emde Boas memory layout

- Example :



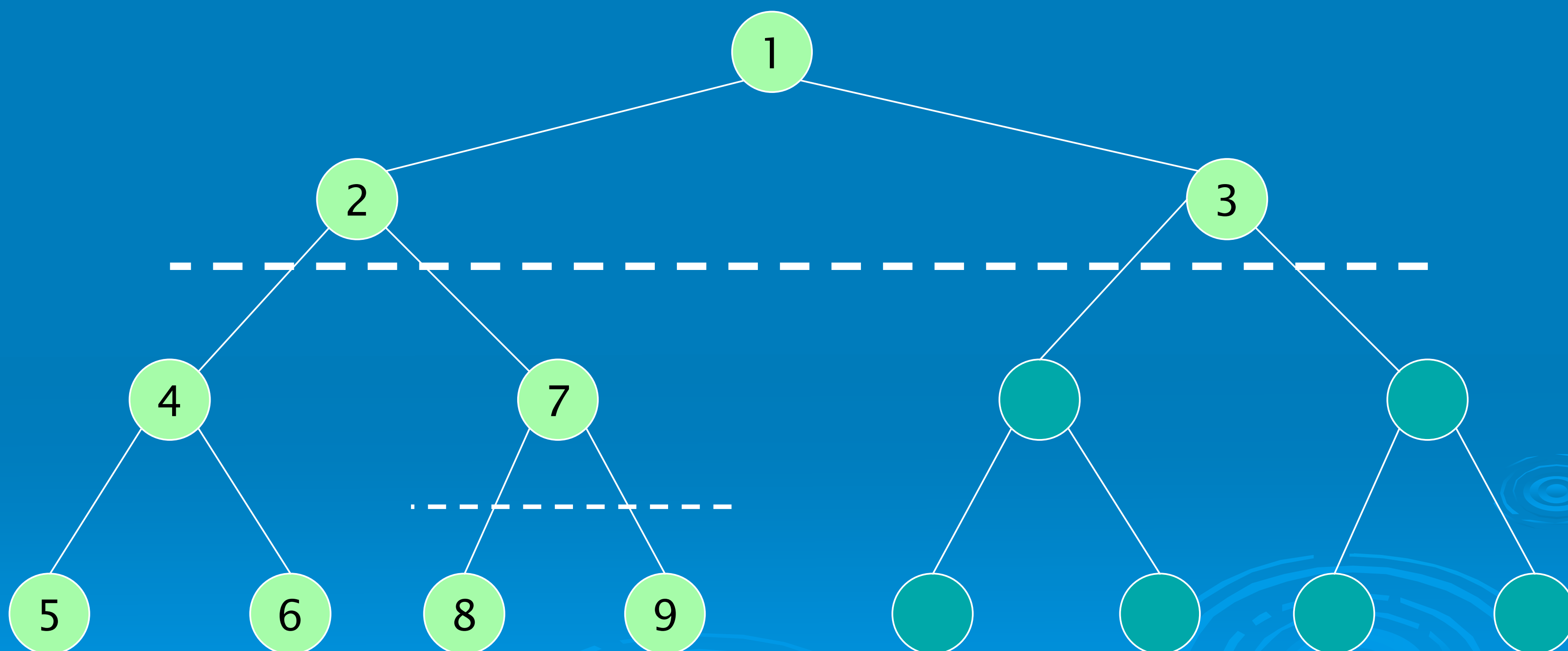
van Emde Boas memory layout

- Example :



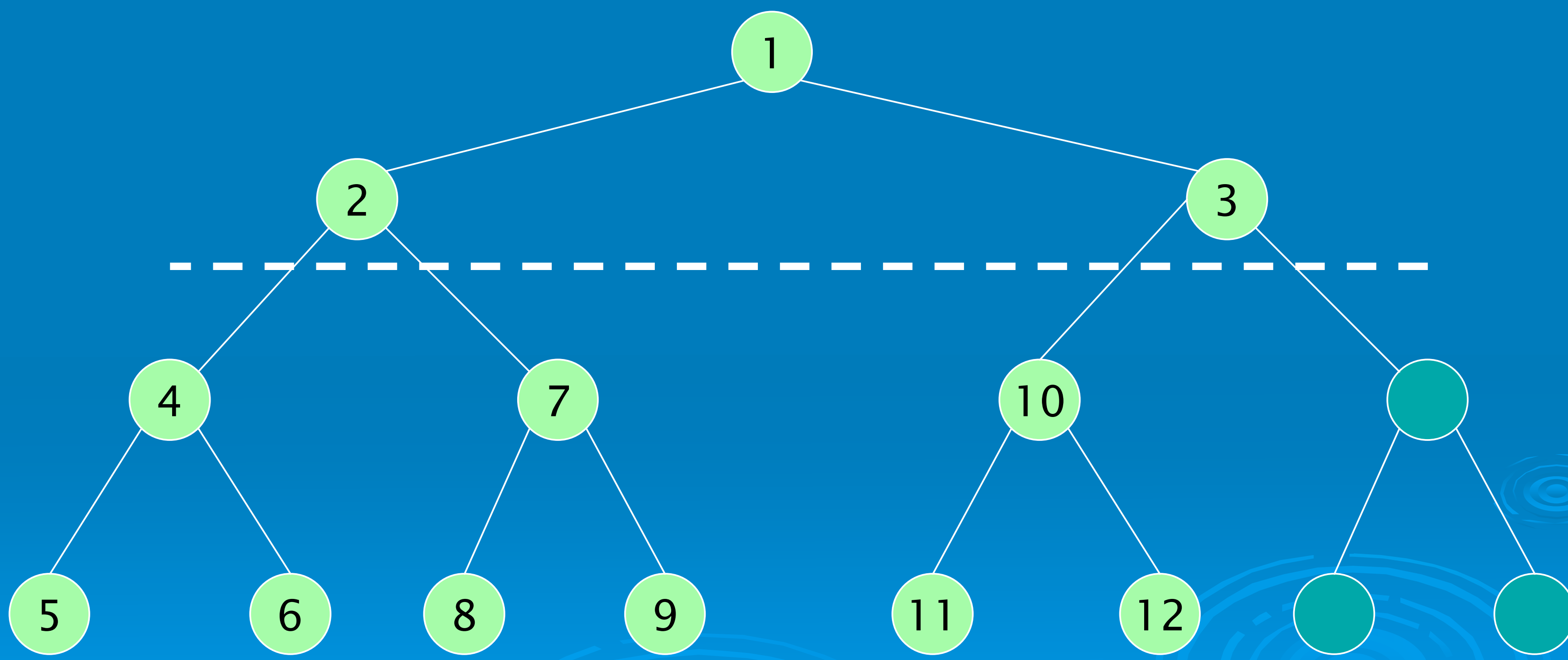
van Emde Boas memory layout

- Example :



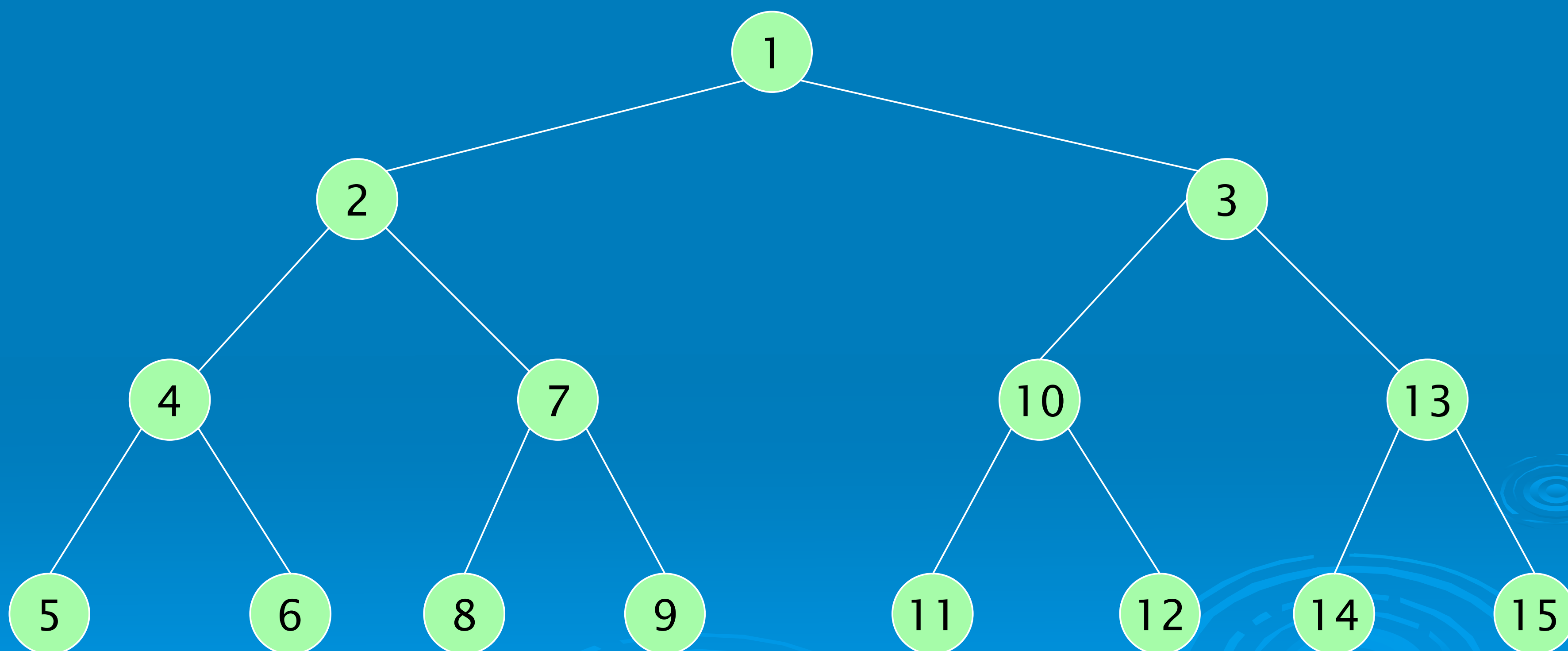
van Emde Boas memory layout

- Example :



van Emde Boas memory layout

- Example :



The algorithm

- Search:
  - Standard search in a binary tree.
  - Memory transfers: $O(\log_B n)$ worst case
- Range query:
  - Standard range query in a binary tree:
    - Search the smallest element in the range
    - Make an inorder traversals till you reach an element greater then or equals to the greatest element in the range.
  - Memory transfers: $O(\log_B n + k/B)$ worst case

The algorithm

- Notations:

$T =$ the dynamic tree.

$H =$ the height of the static complete tree.

$s(v) =$ the size of the subtree in the complete tree rooted at v.

$$\rho(v) = \frac{|T_v|}{s(v)} =$$ the density of v.

The algorithm

We'll define a sequence of evenly distributedv density thresholds: $0 < \tau_1 < \tau_2 < \ldots < \tau_H = 1$

by : $\tau_i = \tau_{i-1} + \Delta$

$$\Delta = \frac{(1 - \tau_1)}{(H - 1)}$$

Example: $H = 5$

$$\tau_1 = 0.6$$
$$\tau_2 = 0.7$$
$$\tau_3 = 0.8 \qquad \Longrightarrow \qquad \Delta = 0.1$$
$$\tau_4 = 0.9$$
$$\tau_5 = 1$$

The algorithm

Invariant: for the root r:

$$\rho(r) \leq \tau_1 \;\Rightarrow\; H \geq \log\left(\frac{n}{\tau_1} + 1\right)$$

- Insertions:
  - Locate the position in T of the new node (regular search)
  - If d(v) = H+1 we rebalance T

Rebalancing

1. Find the nearest ancestor w of v with:

$$\rho(w) < \tau_{d(w)}$$

- That means: find the nearest ancestor of v which is not too dense.
- In order to do so we have to calculate $|T_w|$
- We can do it by a simple traversal – Why?

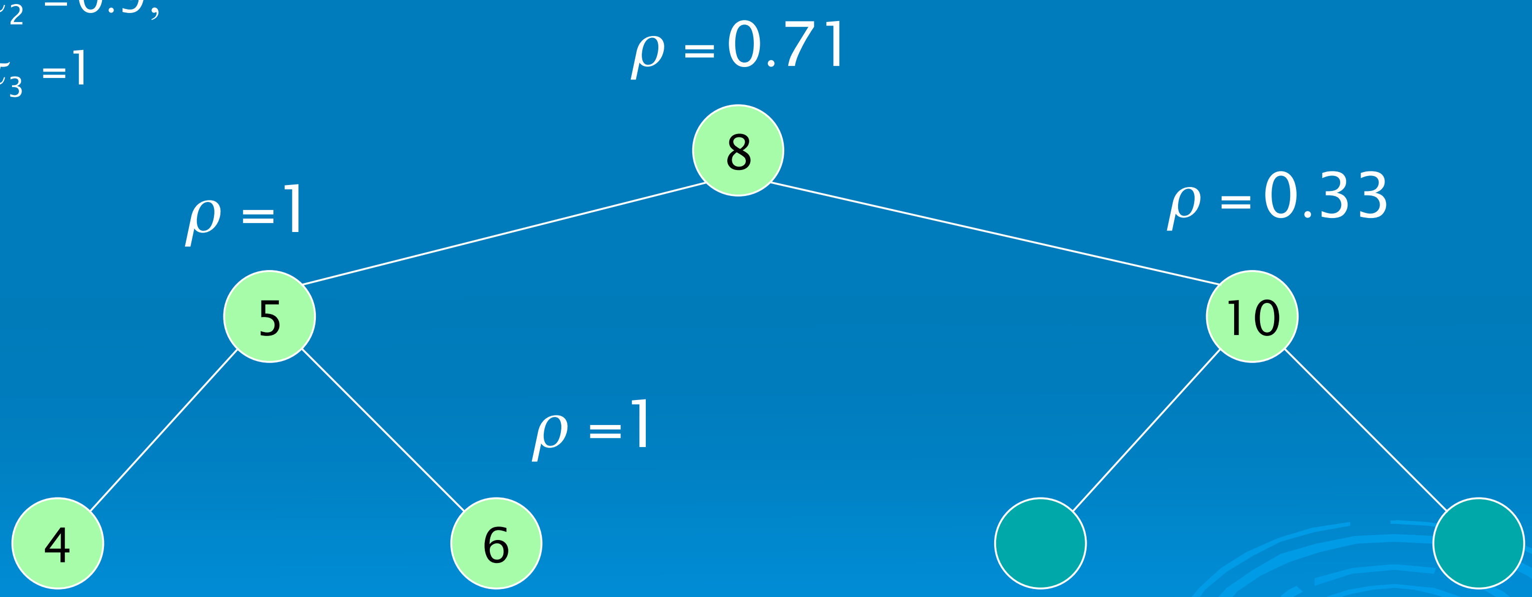
Rebalancing

2. After having located w we rebalance w as follows:
   - Create a sorted array of all elements in T_w by an inorder traversal of T_w.
   - The middle element in the array stored at w.
   - The smallest (greatest) half elements are recursively redistributed at the left (right) child of w.

Rebalancing

- Example : insert 7

$\tau_1 = 0.8,$
$\tau_2 = 0.9,$
$\tau_3 = 1$



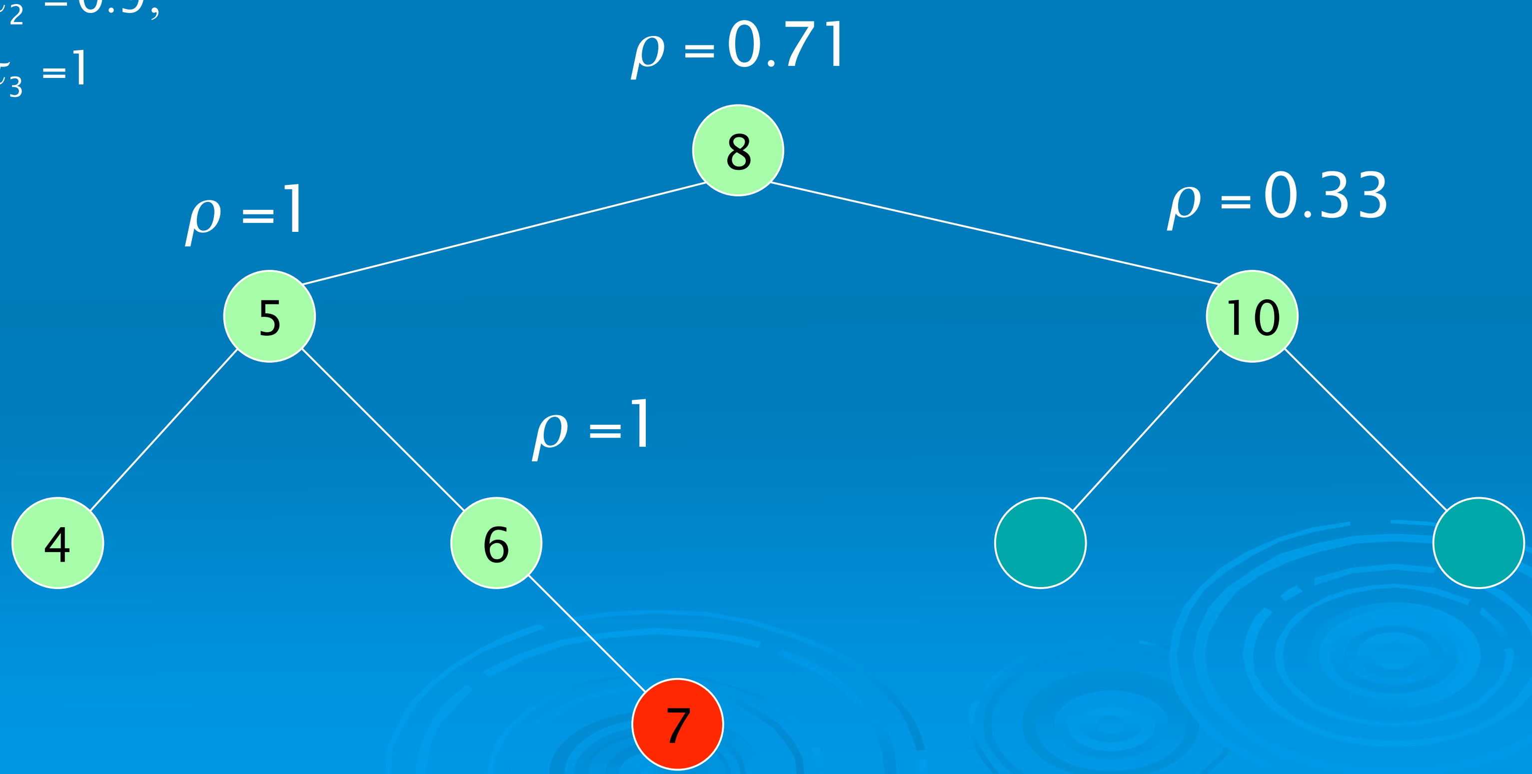
Rebalancing

- Example : insert 7

$\tau_1 = 0.8,$

$\tau_2 = 0.9,$

$\tau_3 = 1$

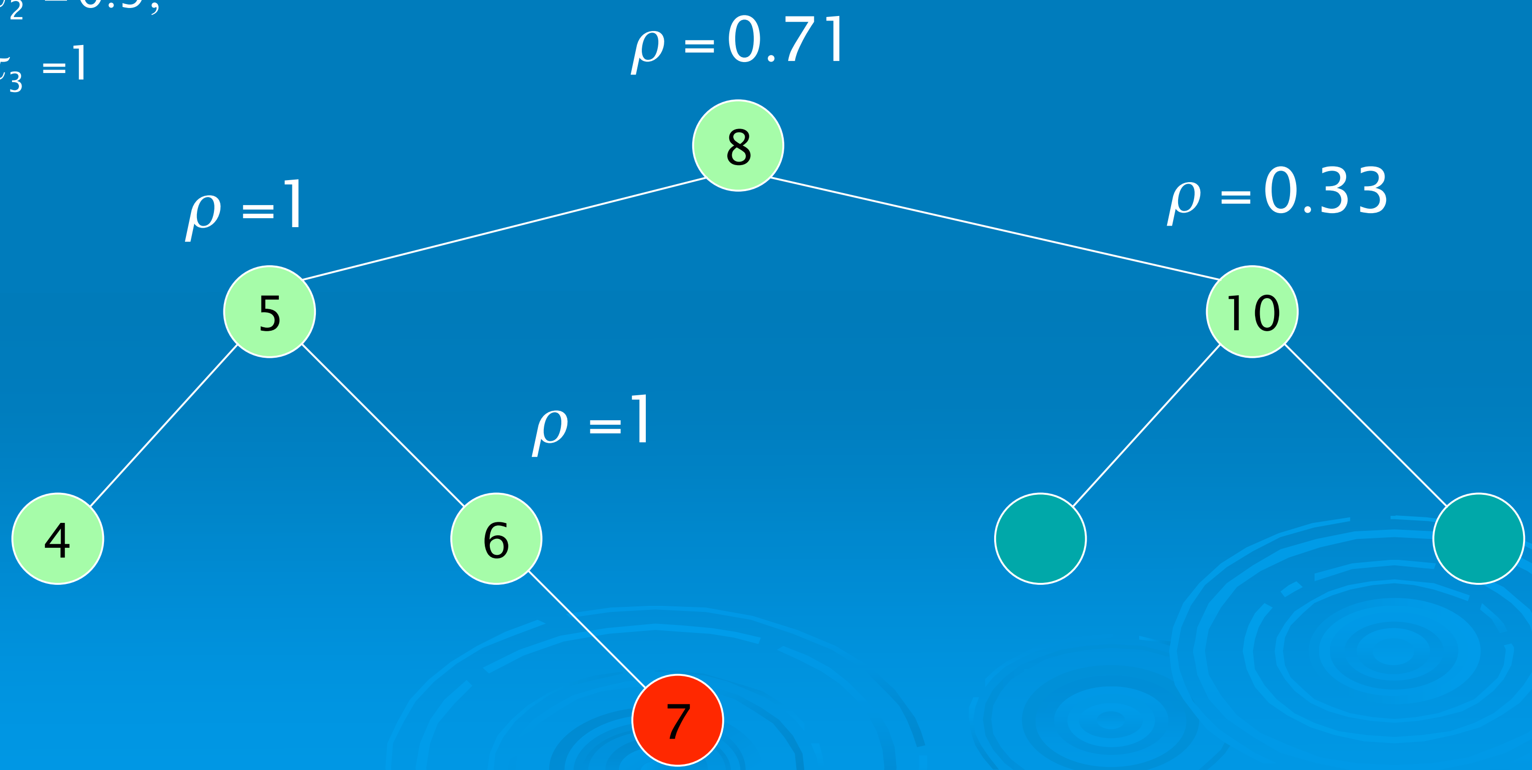


Rebalancing

- Example : insert 7

$\tau_1 = 0.8,$
$\tau_2 = 0.9,$
$\tau_3 = 1$

4	5	6	7	8	10



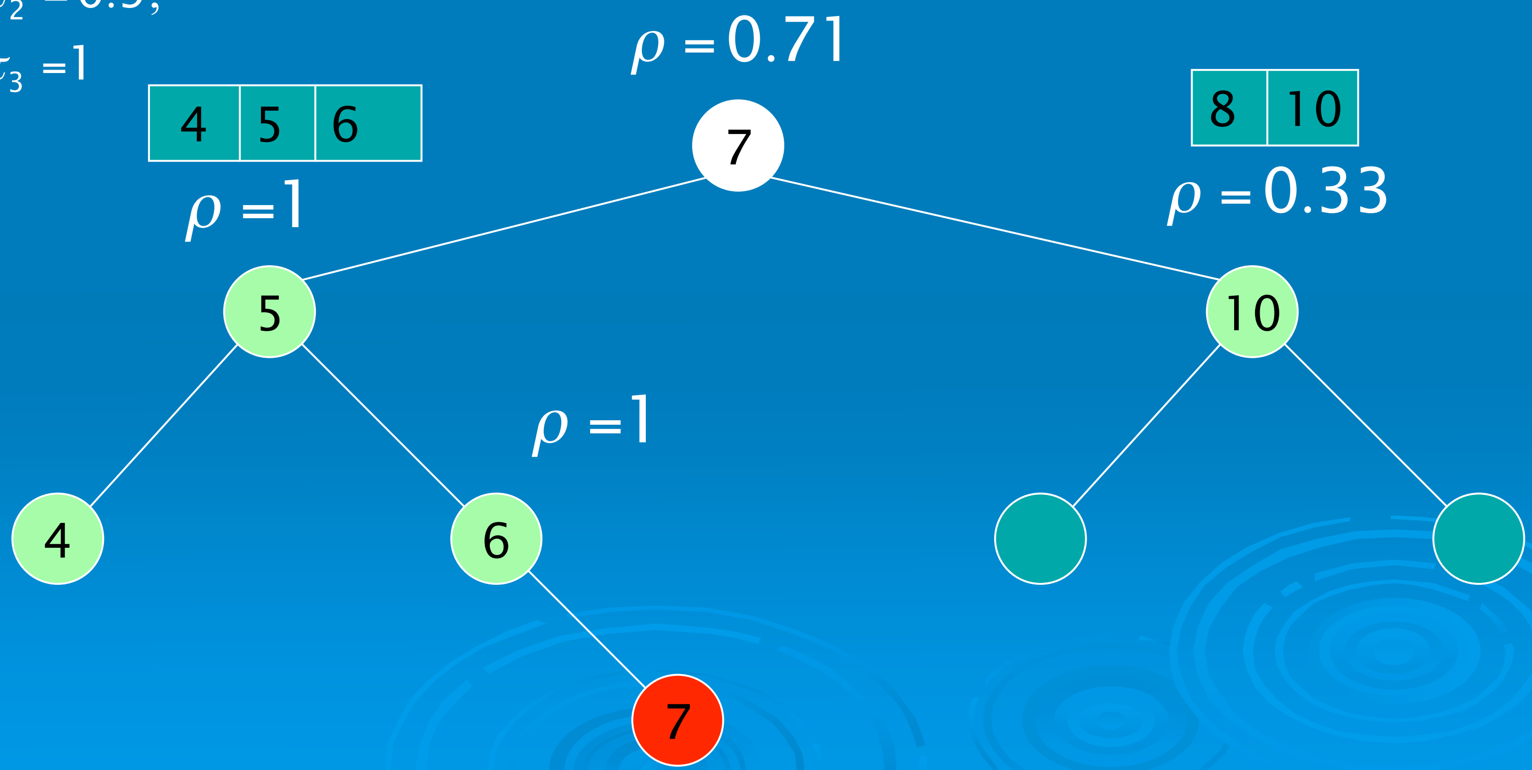
Rebalancing

- Example : insert 7

$\tau_1 = 0.8,$

$\tau_2 = 0.9,$

$\tau_3 = 1$



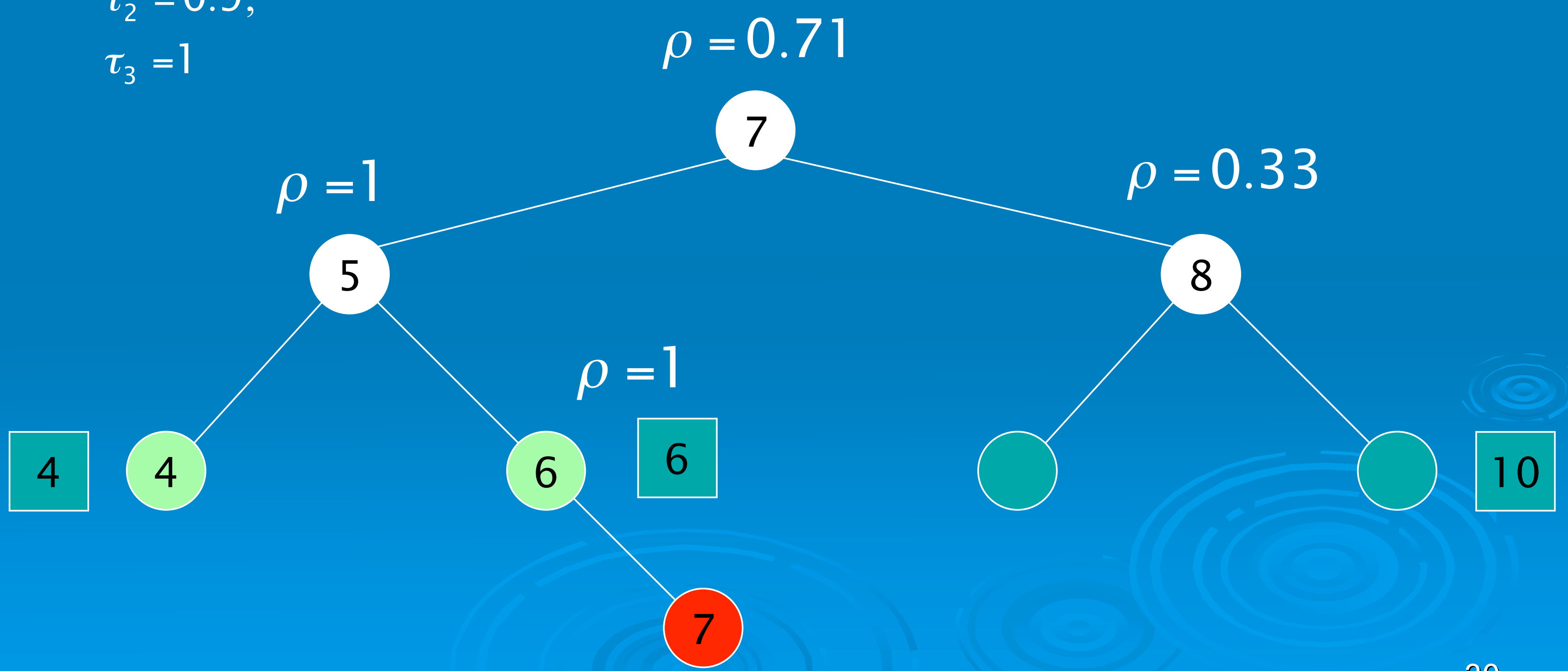
Rebalancing

- Example : insert 7

$\tau_1 = 0.8,$

$\tau_2 = 0.9,$

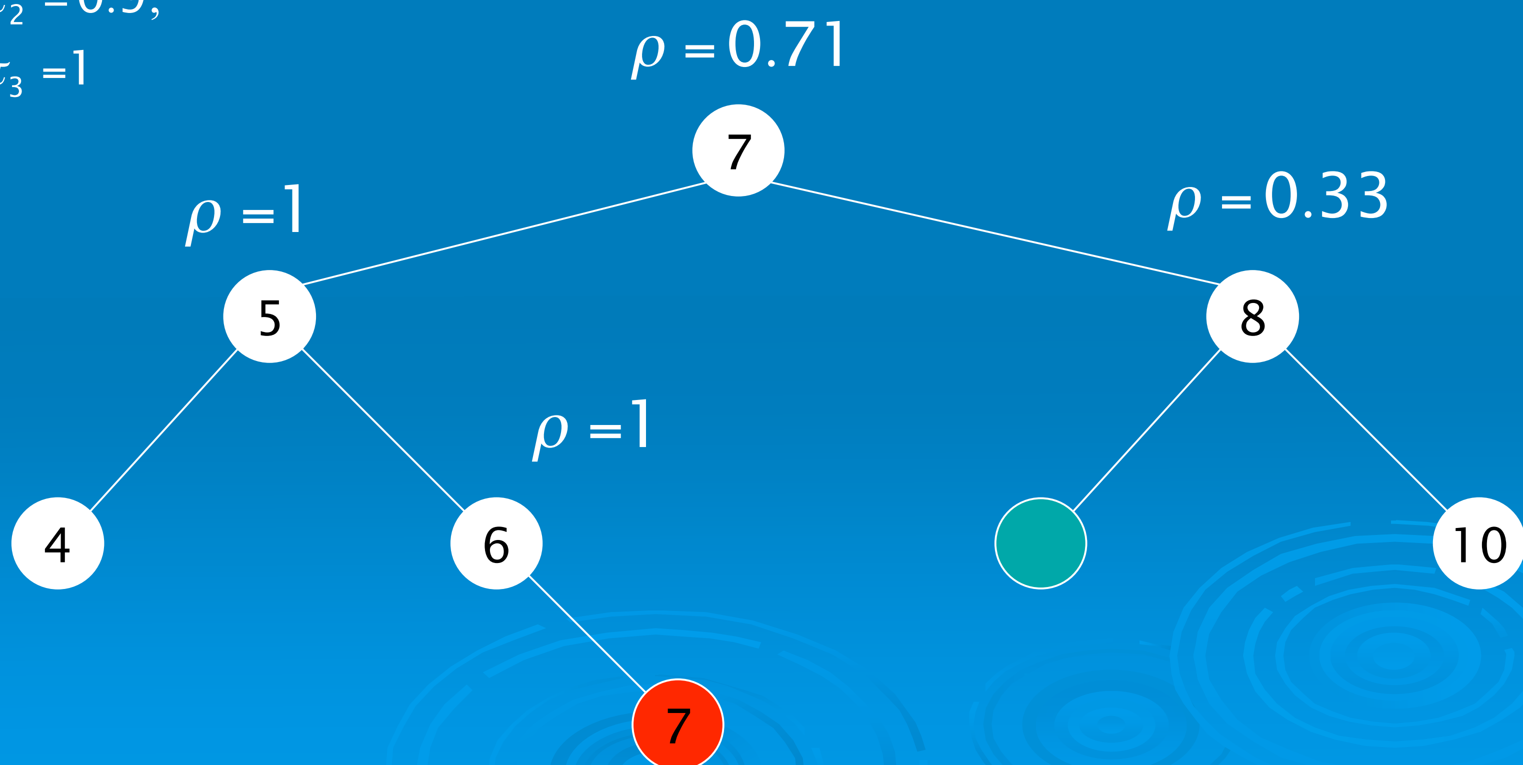
$\tau_3 = 1$



Rebalancing

- Example : insert 7

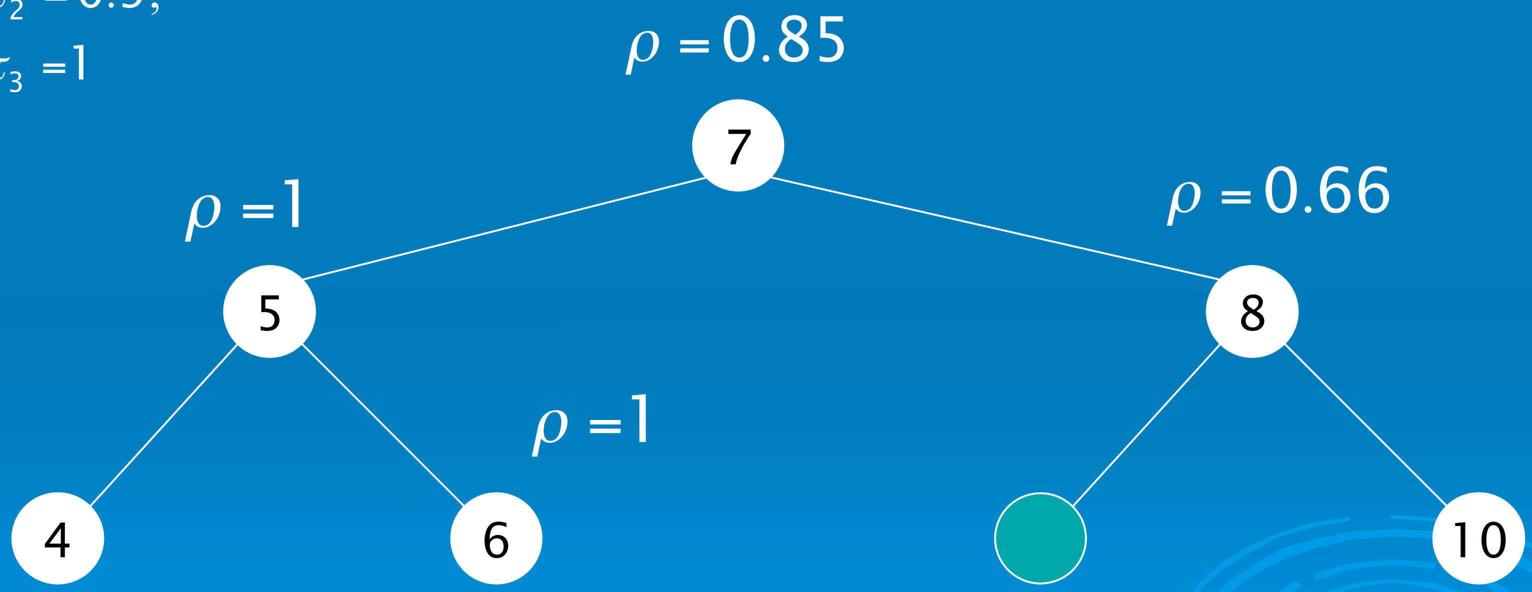
$\tau_1 = 0.8,$
$\tau_2 = 0.9,$
$\tau_3 = 1$



Rebalancing

- Example : insert 7

$\tau_1 = 0.8,$
$\tau_2 = 0.9,$
$\tau_3 = 1$



- The next insertion will cause a rebuilding

Insertion complexity

- Lemma: A redistribution at v implies

$$\lfloor \rho(v)\cdot s(w)-1 \rfloor \le |T_w| \le \lceil \rho(v)\cdot s(w) \rceil$$

  for all descendants w of v
- In other words: after a redistribution at v, for all descendant w

$$\rho(w) \cong \rho(v)$$

- Proof: (induction)

Insertion complexity

- Theorem:

Insertions require amortized $O(\frac{\log^2 n}{1-\tau_1})$ time

and amortized $O(\log_B n + \frac{\log^2 n}{B(1-\tau_1)})$ memory transfers

Insertion complexity (time)

- Proof:
  - Consider a distribution at a node v, caused by an insertion below v.
  - $\Longrightarrow$ for a child w of v :

$$\rho(w) \geq \tau_{d(w)} \;\Rightarrow\; \left|T_w\right| \geq \tau_{d(w)} \cdot s(w)$$

Insertion complexity (time)

- Proof (cont.):
  - The Lemma argues that immediately after a redistribution at v, for all descendant w of v:

$$\left|T_w\right| \le \left\lceil \rho(v) \cdot s(w) \right\rceil$$

  - Since the redistribution took place at v:

$$\rho(v) < \tau_{d(v)}$$

  - $\Longrightarrow$

$$\left|T_w\right| \le \tau_{d(v)} \cdot s(w) + 1$$

Insertion complexity (time)

- Proof (cont.):
  - It follows that the number of insertions below w since the last redistribution at v or an ancestor of v is at least:

$$\tau_{d(w)} \cdot s(w) - (\tau_{d(v)} \cdot s(w) + 1)$$

The number of elements in w right now, because w become “dense”

The number of elements at w immediately after the last redistribution at v or at ancestor of v

Insertion complexity (time)

- Proof (cont.):

$$\tau_{d(w)} \cdot s(w) - (\tau_{d(v)} \cdot s(w) + 1)$$

$$= \tau_{d(w)} \cdot s(w) - \tau_{d(v)} \cdot s(w) - 1$$

$$= s(w)\left(\tau_{d(w)} - \tau_{d(v)}\right) - 1$$

$$= s(w) \cdot \Delta - 1$$

Insertion complexity (time)

- Proof (cont.):
  - The redistribution at v takes O(s(v)) which can be covered by $\max\{1, s(w) \cdot \Delta - 1\}$ elements. Hence, each node is charged at:

$$O\left(\frac{s(v)}{\max\{1, s(w) \cdot \Delta - 1\}}\right) = O\left(\frac{1}{\Delta}\right)$$

    for each of the mentioned insertions below w.

Insertion complexity (time)

- Proof (cont.):
  - Since each node has at most H ancestors it will be charged at most H times and the amortized complexity will be:

$$H \cdot O\left(\frac{1}{\Delta}\right) = O\left(\frac{H}{\Delta}\right) = O\left(\frac{H^2}{1-\tau_1}\right) = O\left(\frac{\log^2 n}{1-\tau_1}\right)$$

Insertion complexity (memory transfers)

- Proof:
  - A top down search requires $O(\log_B n)$ memory transfers.
  - The redistribution itself is like a range query when k=s(v), hence takes: O([illegible] B }) mem[illegible]
  - Amortized [illegible] got:

Finding the ancestor

Redistribution at the ancestor

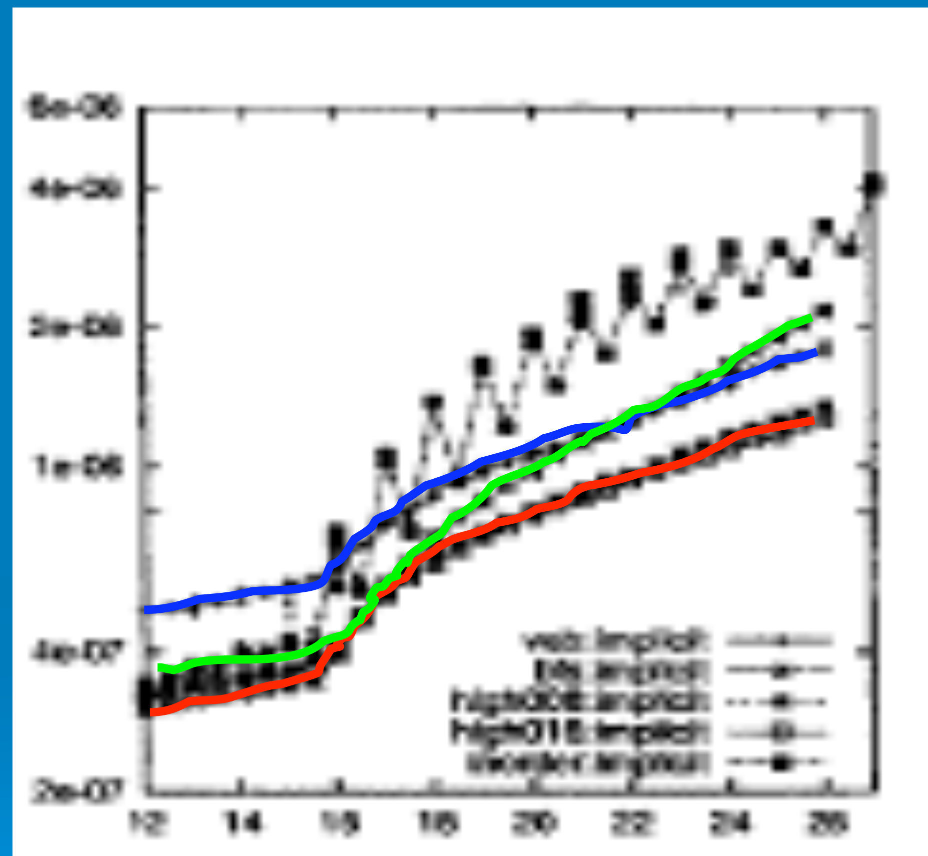
$$O(\log_B n + \frac{\log^2 n}{B(1-\tau_1)})$$

Experiments & Results

- The platform:
  - Two 1[GHz] Pentium 3 processors.
  - 256[KB] of cache
  - 1[GB] R.A.M.
  - Operating system: Linux kernel 2.4.3-12smp.
- The experiments – searches with:
  - Various tree sizes (different n)
  - Various memory layouts
  - Implicit and pointer based versions

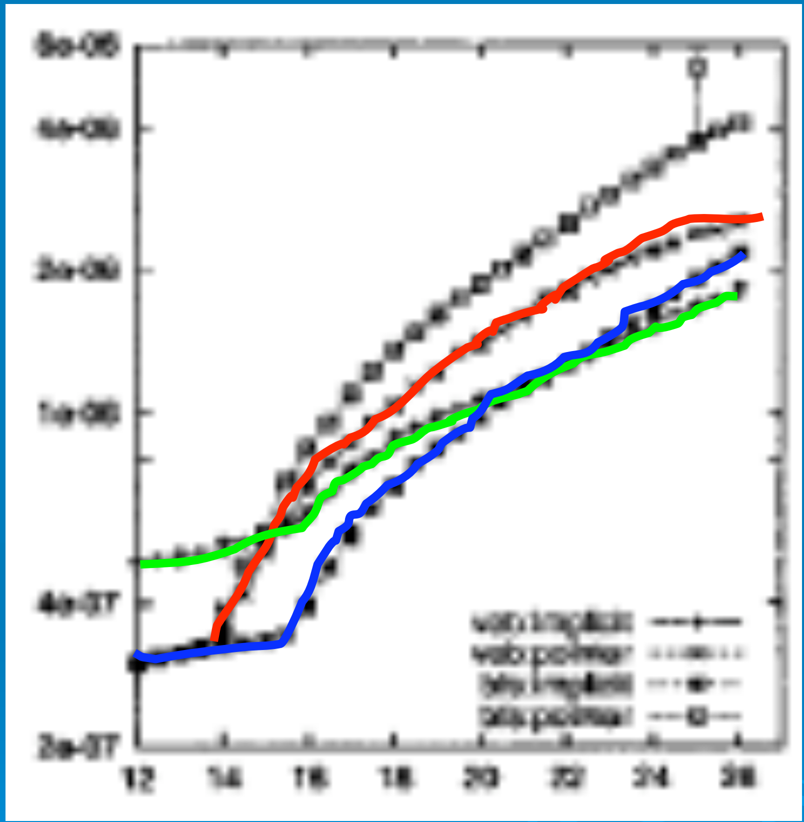
Experiments & Results

- Red: cache aware algs.
- Blue: v.E.B layout.
- Green: BFS implicit
- Black: inorder implicit



Experiments & Results

- Red: v.E.B pointer
- Green: v.E.B implicit
- Black: B.F.S. pointer
- Blue: B.F.S. implicit



Summary

- We introduced the van Emde Boas layout.
- We learned a cache oblivious search tree implementation and its time & memory transfers bounds.
- We saw that this implementation is indeed efficient and competitive with cache aware implementations.

Thank you all

14/6/04

Elad Levi