Memory Optimization

Some slides from Christer Ericson

Sony Computer Entertainment, Santa Monica

Overview

- We have seen how to reorganize matrix computations to improve temporal and spatial locality
  - Improving spatial locality required knowing the layout of the matrix in memory
- Orthogonal approach
  - Change the representation of the data structure in memory to improve locality for a given pattern of data accesses from the computation
  - Less theory exists for this but some nice results are available for trees: van Emde Boas tree layout
- Similar ideas can be used for graph algorithms as well
  - However there is usually not as much locality in graph algorithms

Data cache optimization

- Compressing data
- Prefetching data into cache
- Cache-conscious data structure layout
  - Tree data structures
- Linearization caching

Prefetching

- **Software prefetching**
  - Not too early – data may be evicted before use
  - Not too late – data not fetched in time for use
  - Greedy
- **Instructions**
  - iA-64: lfetch (line prefetch)
    - Options:
      - Intend to write: begins invalidations in other caches
      - Which level of cache to prefetch into
  - Compilers and programmers can access through intrinsics

Software prefetching

```
// Loop through and process all 4n elements
for (int i = 0; i < 4 * n; i++)
    Process(elem[i]);
```

```
const int kLookAhead = 4; // Some elements ahead
for (int i = 0; i < 4 * n; i += 4) {
    Prefetch(elem[i + kLookAhead]);
    Process(elem[i + 0]);
    Process(elem[i + 1]);
    Process(elem[i + 2]);
    Process(elem[i + 3]);
}
```

Greedy prefetching

```
void PreorderTraversal(Node *pNode) {
    // Greedily prefetch left traversal path
    Prefetch(pNode->left);
    // Process the current node
    Process(pNode);
    // Greedily prefetch right traversal path
    Prefetch(pNode->right);
    // Recursively visit left then right subtree
    PreorderTraversal(pNode->left);
    PreorderTraversal(pNode->right);
}
```

Data structure representation

- **Cache-conscious layout**
  - Node layout
    - Field reordering (usually grouped conceptually)
    - Hot/cold splitting
  - Overall data structure layout
- **Little compiler support**
  - Easier for non-pointer languages (Java)
  - C/C++: do it yourself

Field reordering

```
struct S {
    void *key;
    int count[20];
    S *pNext;
};
```

```
struct S {
    void *key;
    S *pNext;
    int count[20];
};
```

```
void Foo(S *p, void *key, int k) {
    while (p) {
        if (p->key == key) {
            p->count[k]++;
            break;
        }
        p = p->pNext;
    }
}
```

- Likely accessed together so store them together!

Hot/cold splitting

Hot fields:

```
struct S {
    void *key;
    S *pNext;
    S2 *pCold;
};
```

Cold fields:

```
struct S2 {
    int count[10];
};
```

- Split cold fields into a separate structure
- Allocate all 'struct S' from a memory pool
  - Increases coherence

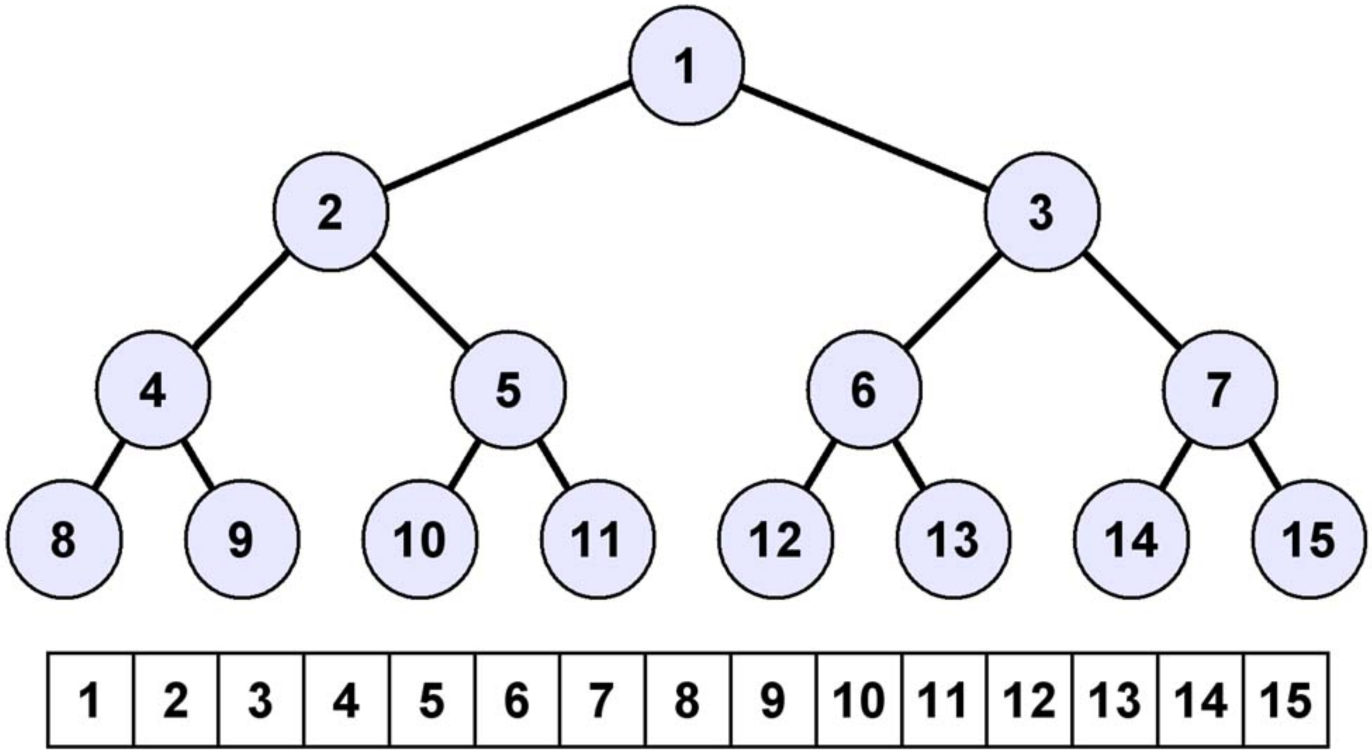
Tree data structures

- **Rearrange nodes**
  - Increase spatial locality
  - Cache-aware vs. cache-oblivious layouts
- **Reduce size**
  - Pointer elimination (using implicit pointers)
  - "Compression"
    - Quantize values
    - Store data relative to parent node

General idea & Working methods

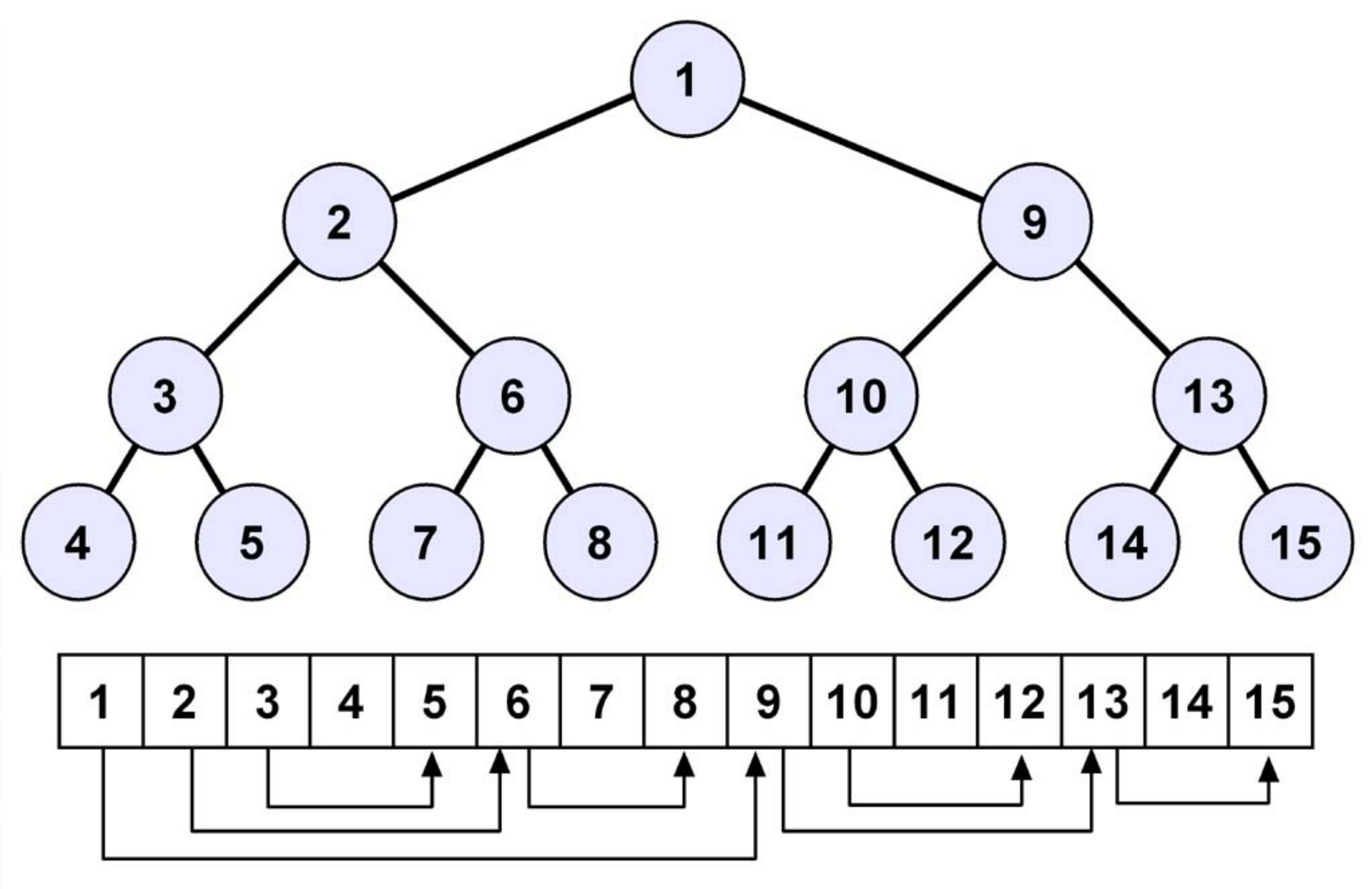
- Definitions:
- A tree T1 can be embedded in another tree T2, if T1 can be obtained from T2 by pruning subtrees.
- Implicit layout - the navigation between a node and its children is done based on address arithmetic, and not on pointers.

Breadth-first order



- Pointer-less: Left(n)=2n, Right(n)=2n+1
- Requires storage for complete tree of height H

Depth-first order



- Left(n) = n + 1, Right(n) = stored index
- Only stores existing nodes

Cache Oblivious Binary Search Trees

Gerth Stolting Brodal
Rolf Fagerberg
Riko Jacob

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

- 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
  - Complete tree permits storing the tree without child pointers
  - However there may be some empty subtrees
- On insertion, create a new static tree of double the size if needed.

General idea & Working methods

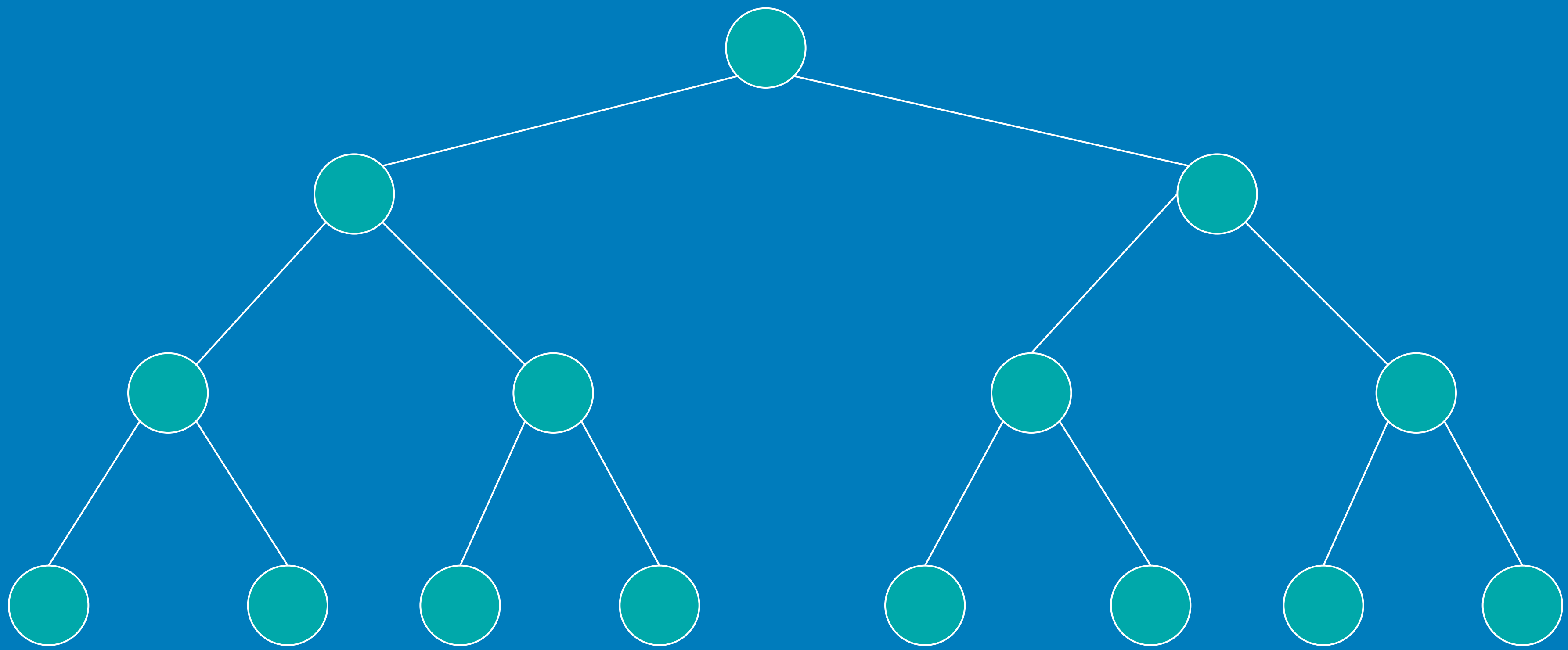
- Advantages:
  - Minimizing memory transfers.
  - Cache obliviousness
  - No pointers – better space utilization:
    - A larger fraction of the structure can reside in lower levels of the memory.
    - More elements can fit in a cache line.
- Disadvantages:
  - Implicit layout: higher instruction count 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 $T_1,\ldots,T_k$

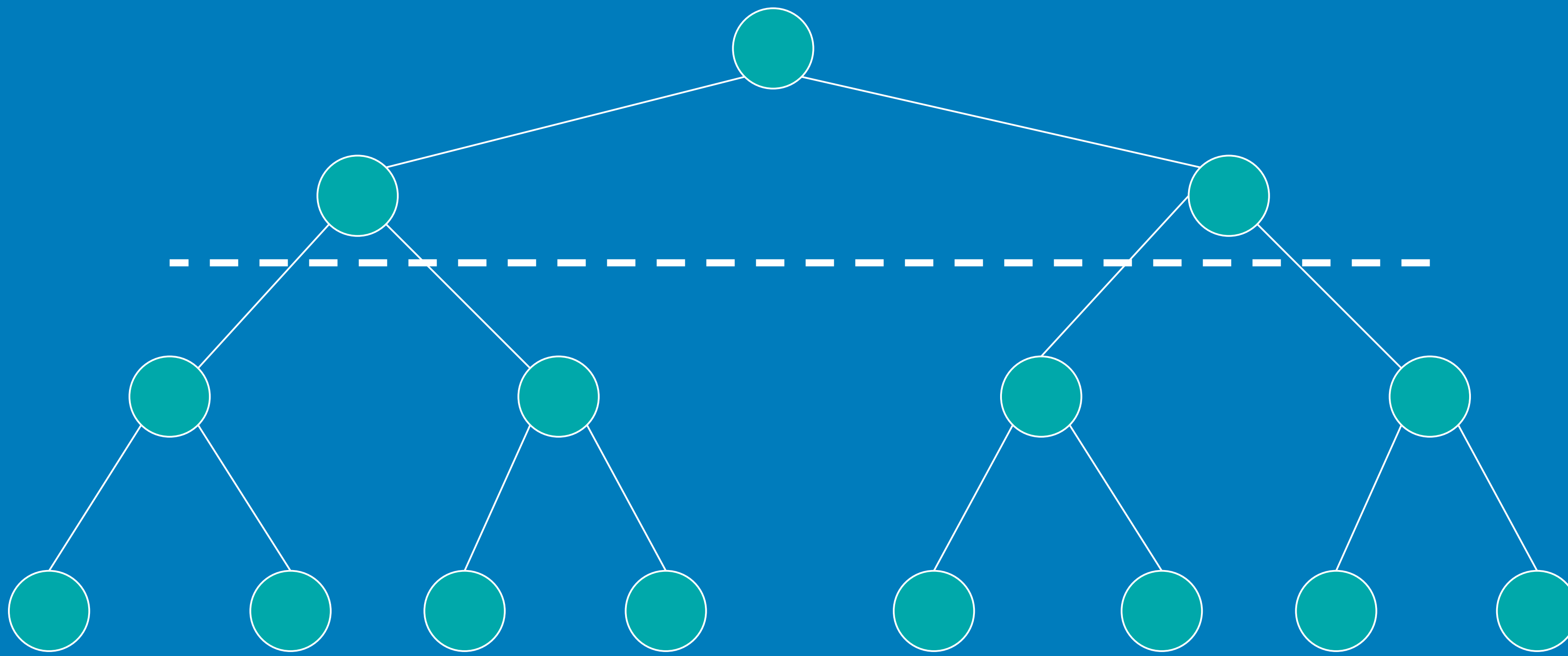
van Emde Boas memory layout

- Example :



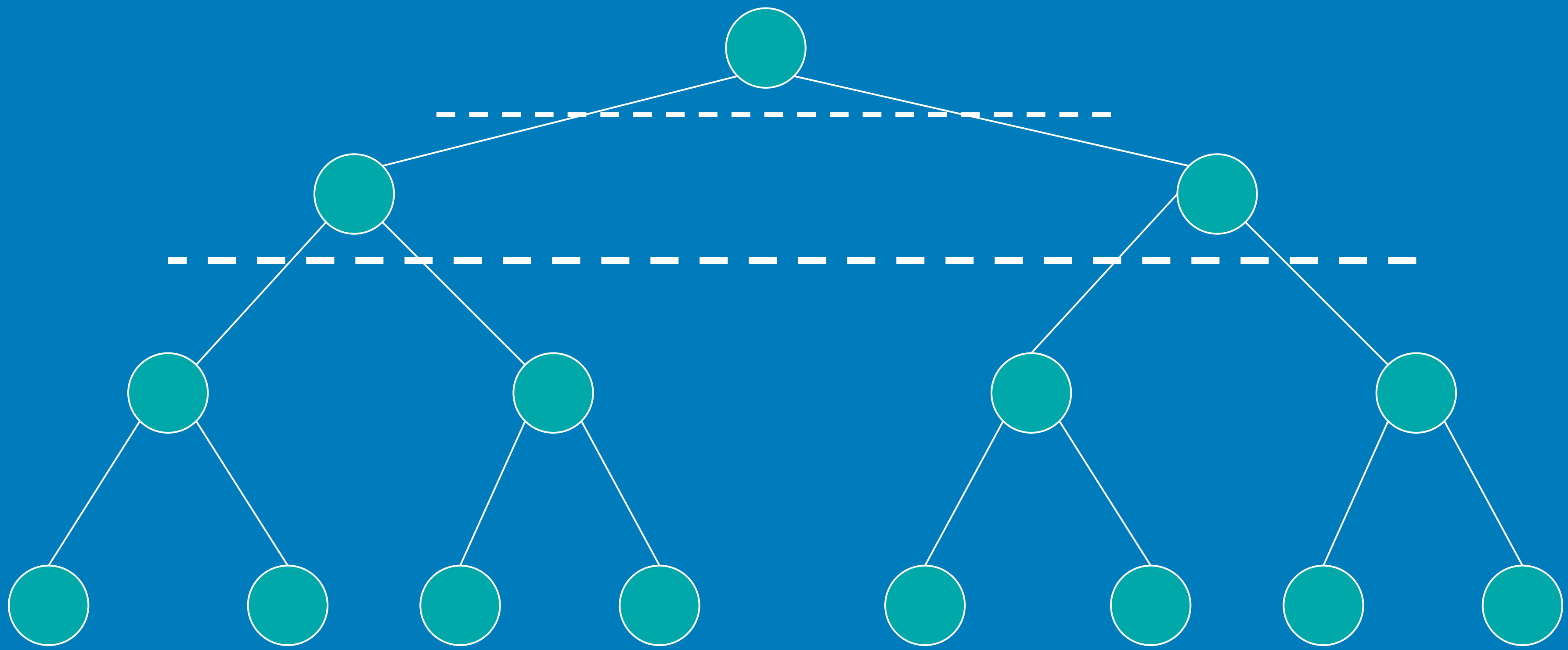
van Emde Boas memory layout

- Example :



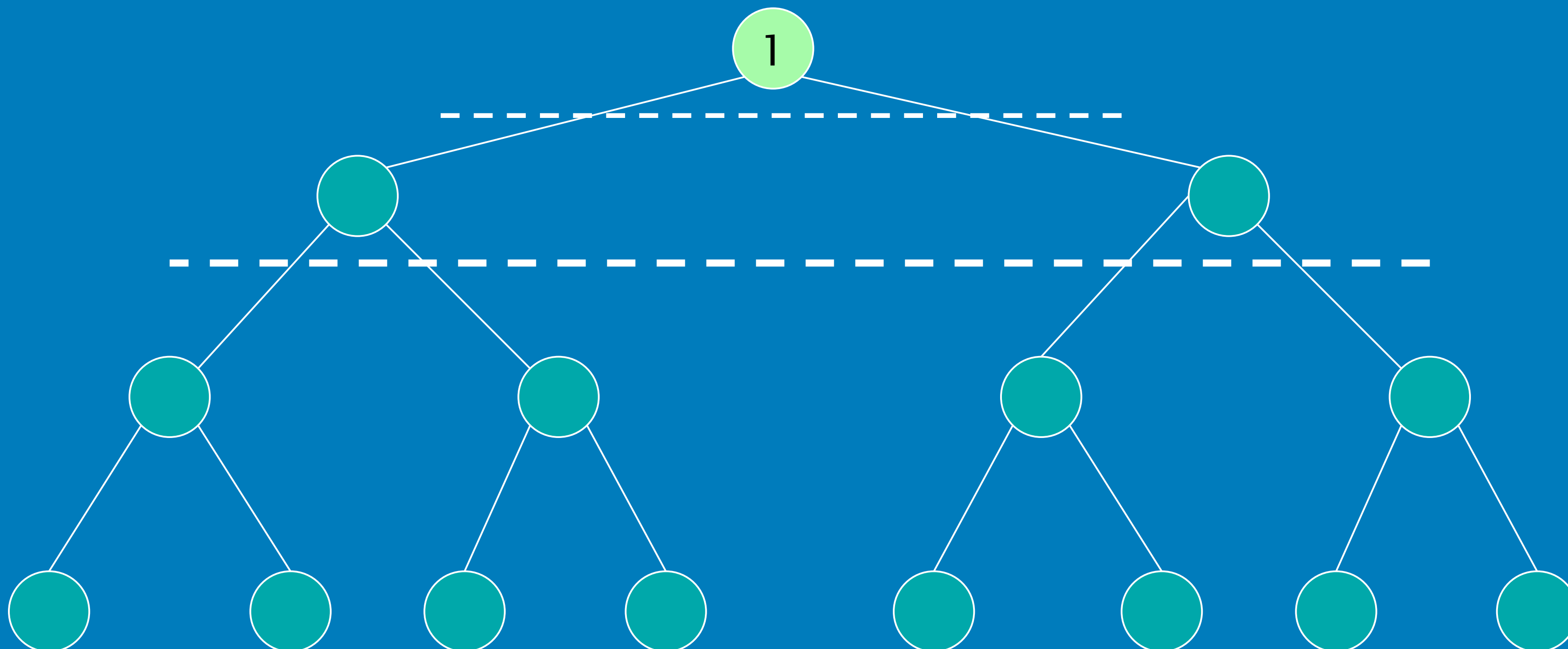
van Emde Boas memory layout

- Example :



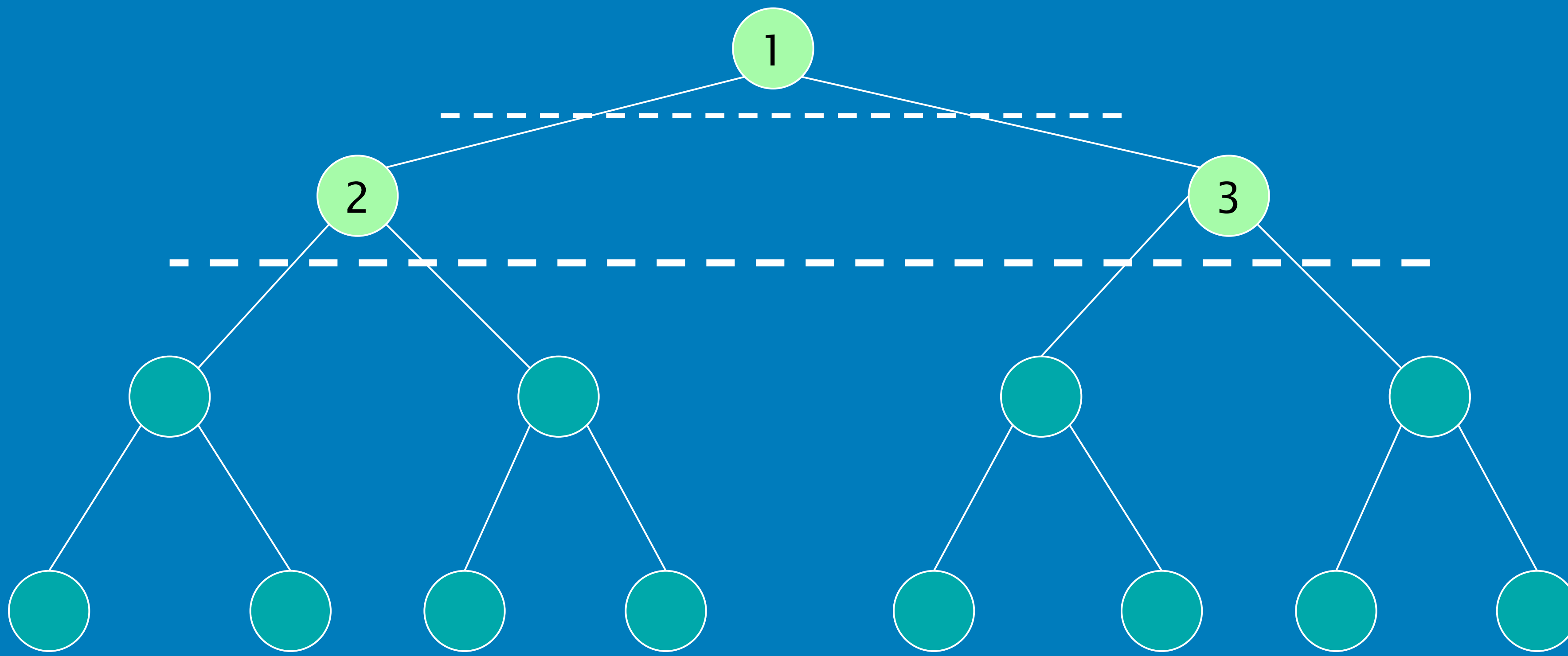
van Emde Boas memory layout

- Example :



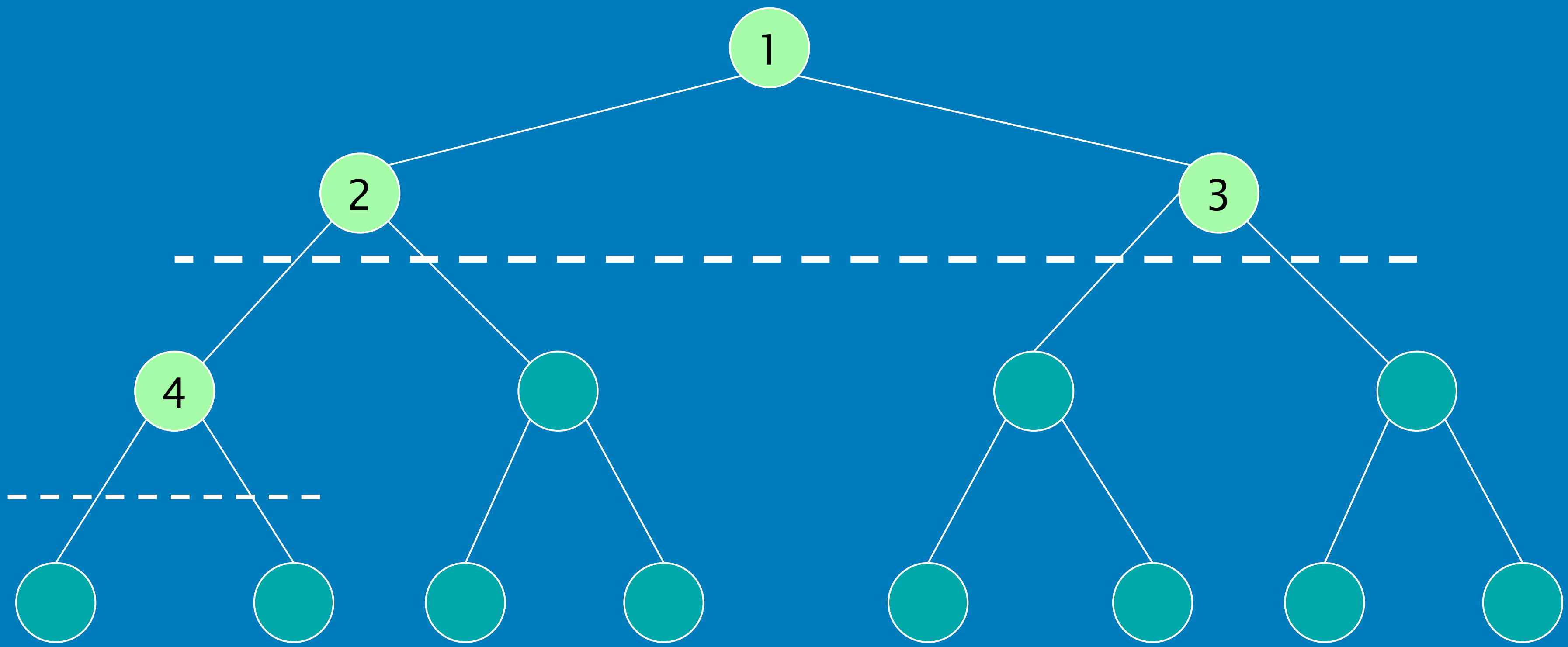
van Emde Boas memory layout

- Example :



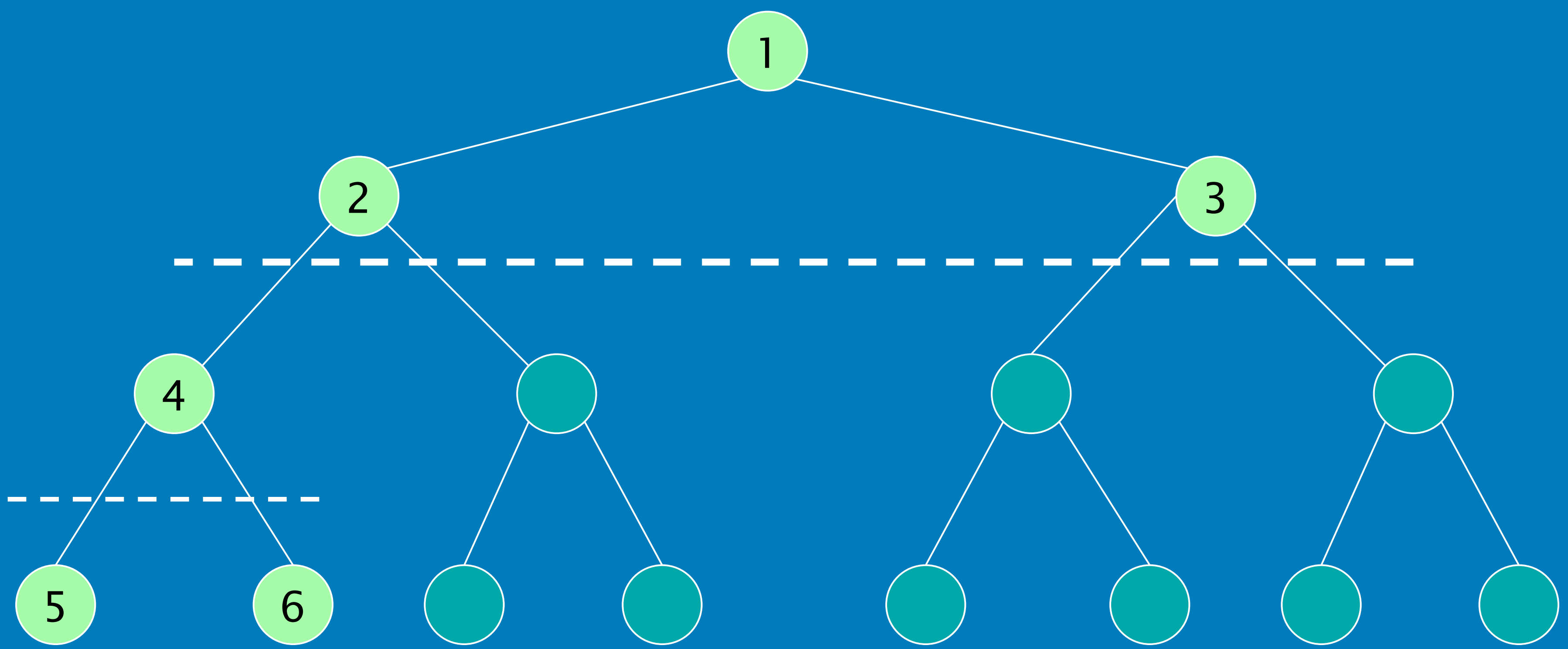
van Emde Boas memory layout

- Example :



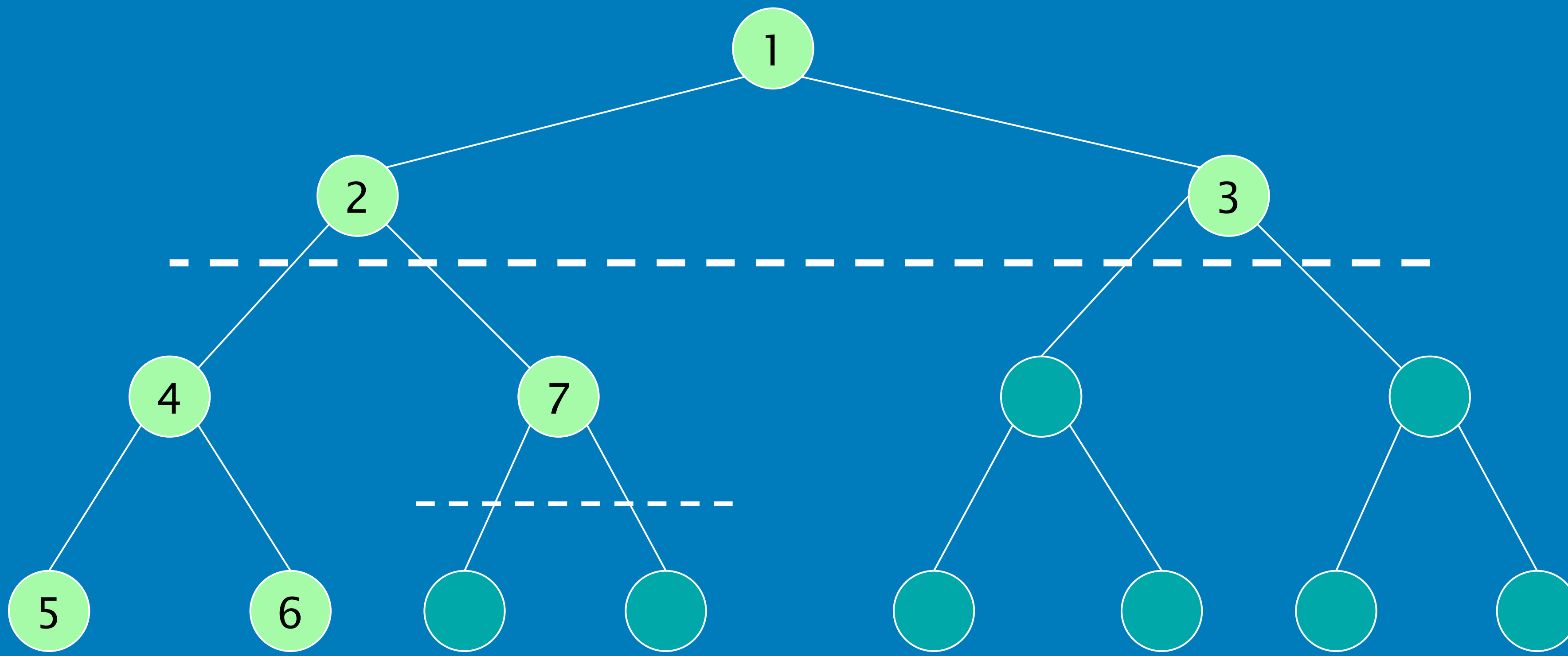
van Emde Boas memory layout

- Example :



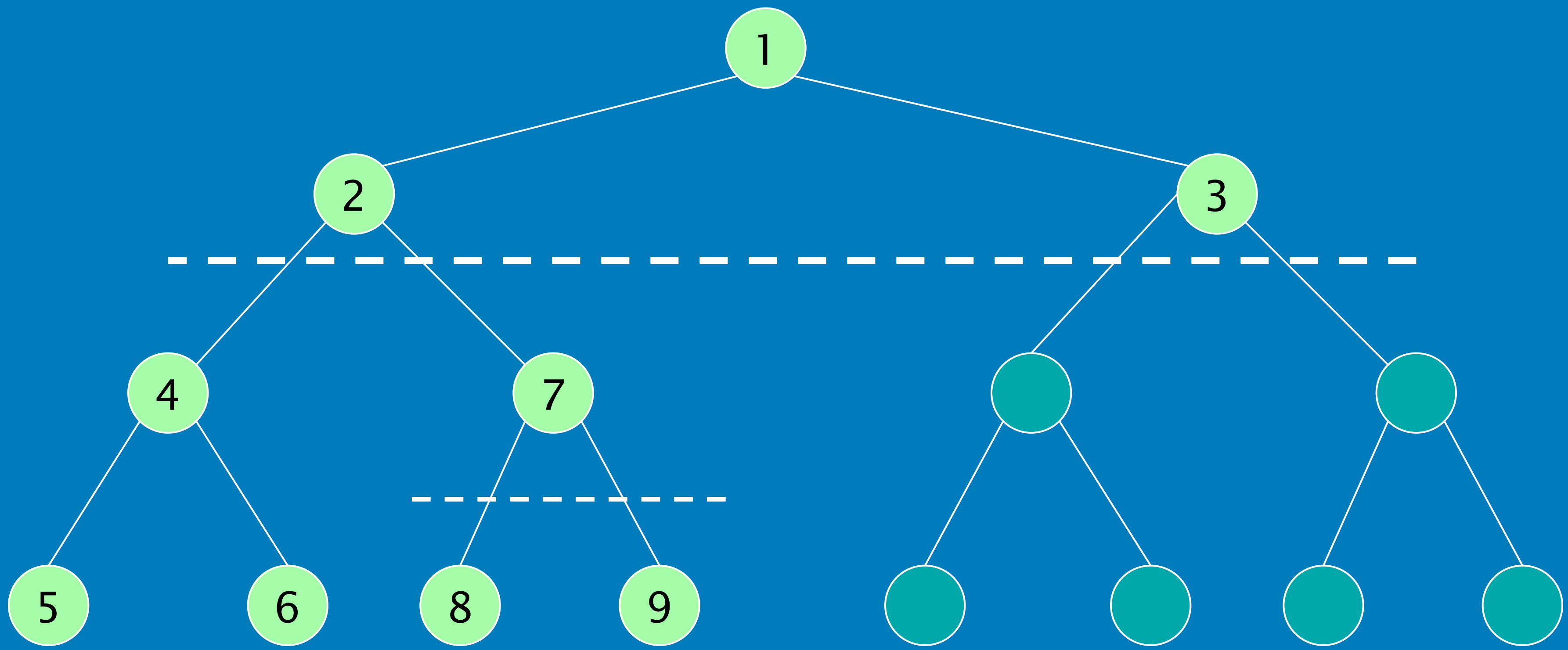
van Emde Boas memory layout

- Example :



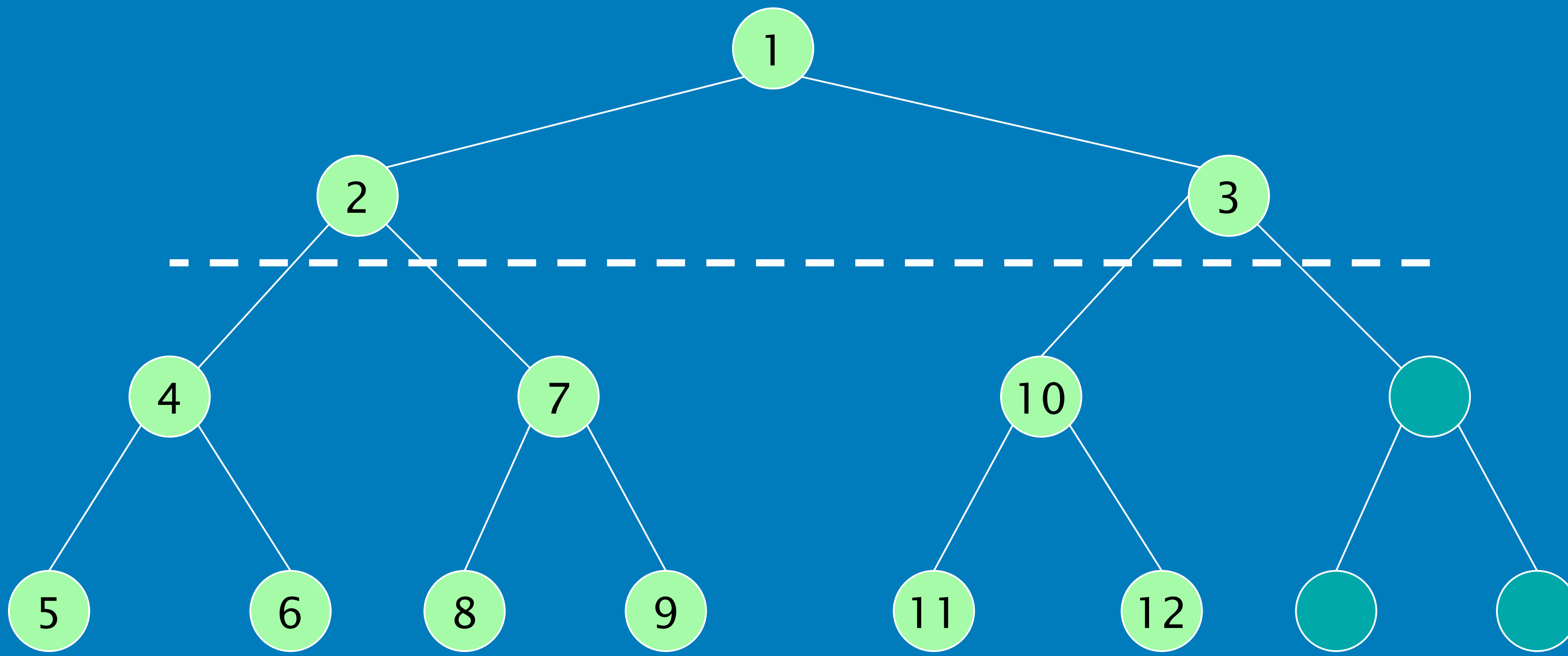
van Emde Boas memory layout

- Example :



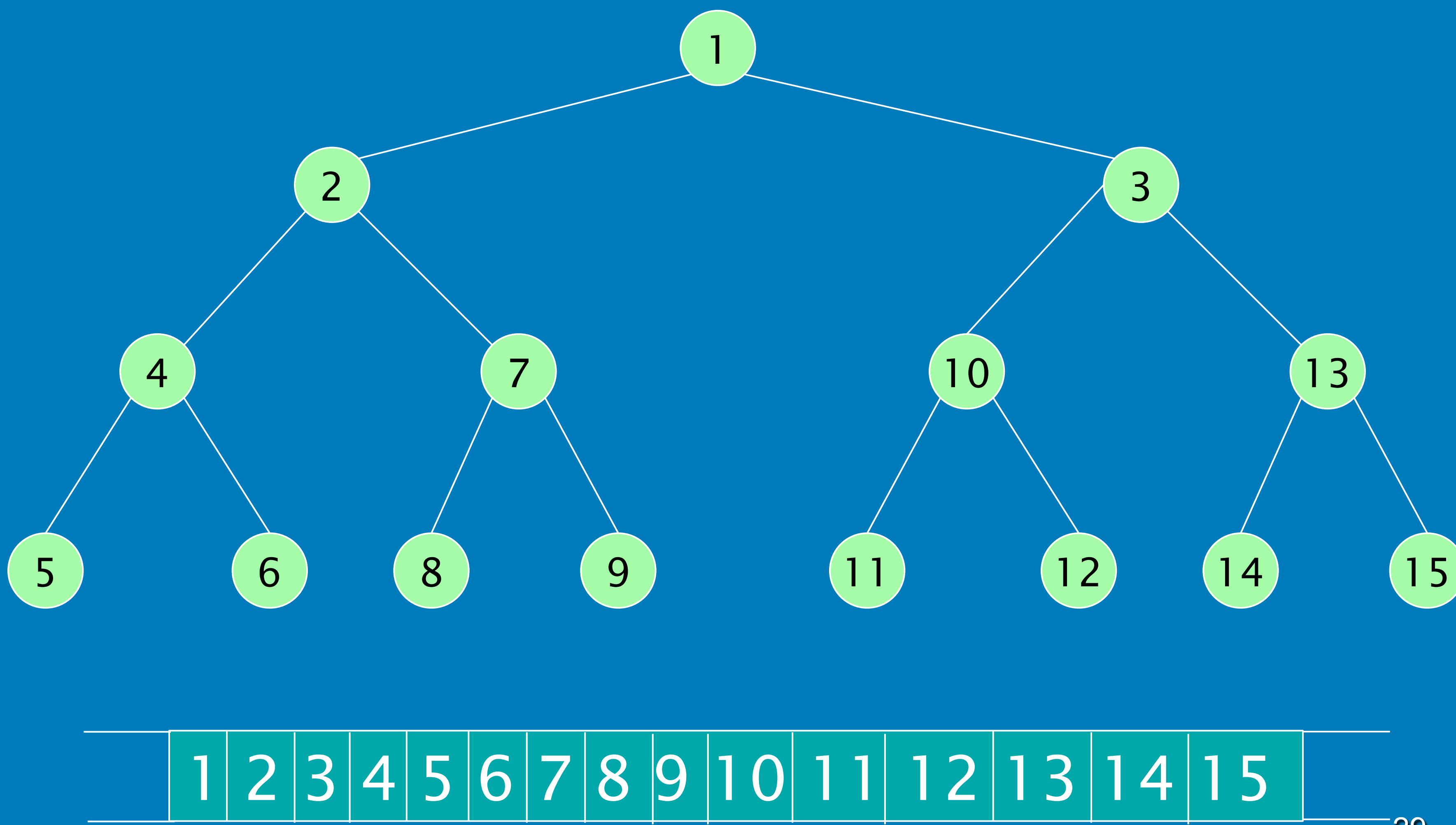
van Emde Boas memory layout

- Example :



van Emde Boas memory layout

- Example :



The algorithm

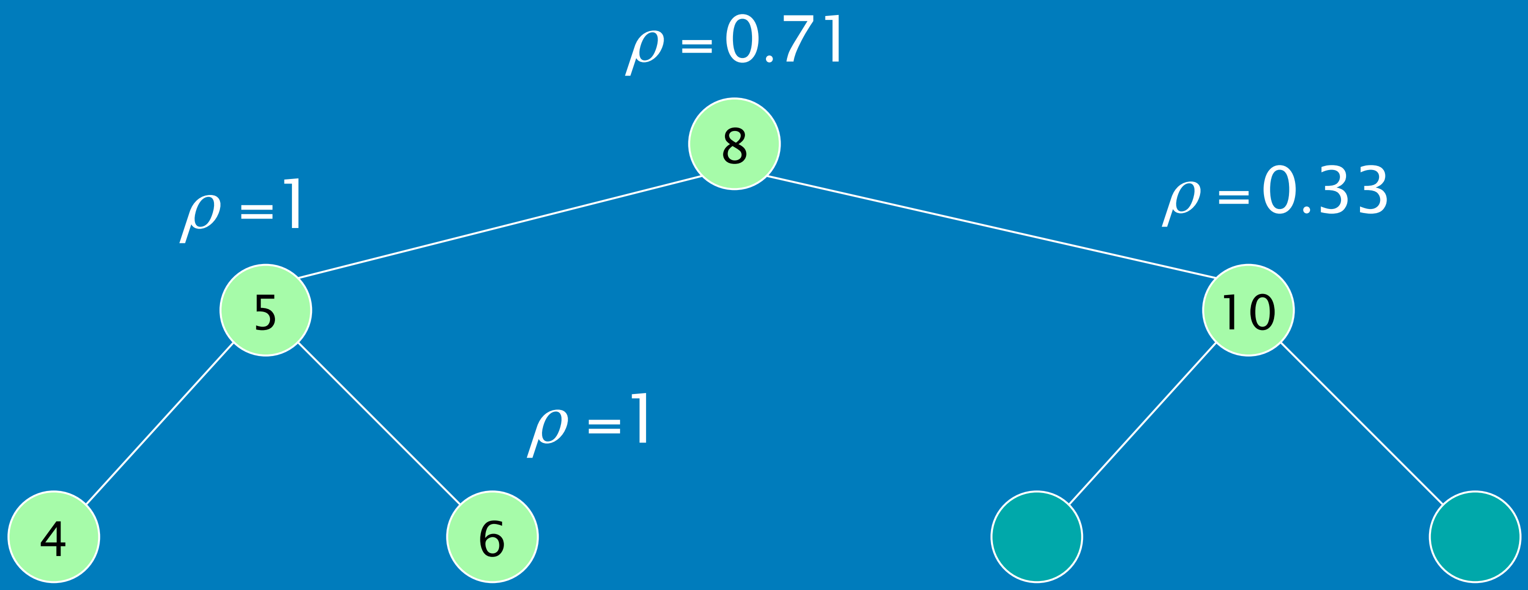
- Search:
  - Standard search in a binary tree.
  - Memory transfers: $O(\log_B n)$ worst case
- Range query:
  - Standard range query in a binary tree:
    - Search the smallest element in the range
    - Make an inorder traversals till you reach an element greater then or equals to the greatest element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

Insertions

- Intuitive idea:
  - Locate the position in T of the new node (regular search)
  - If there is an empty slot there, just insert the new value there
  - If tree has some empty slots, rebalance T and then insert the new value
  - Otherwise, use recursive doubling
    - Allocate a new tree for double the depth of the current tree
    - Copy over values from new tree to old tree

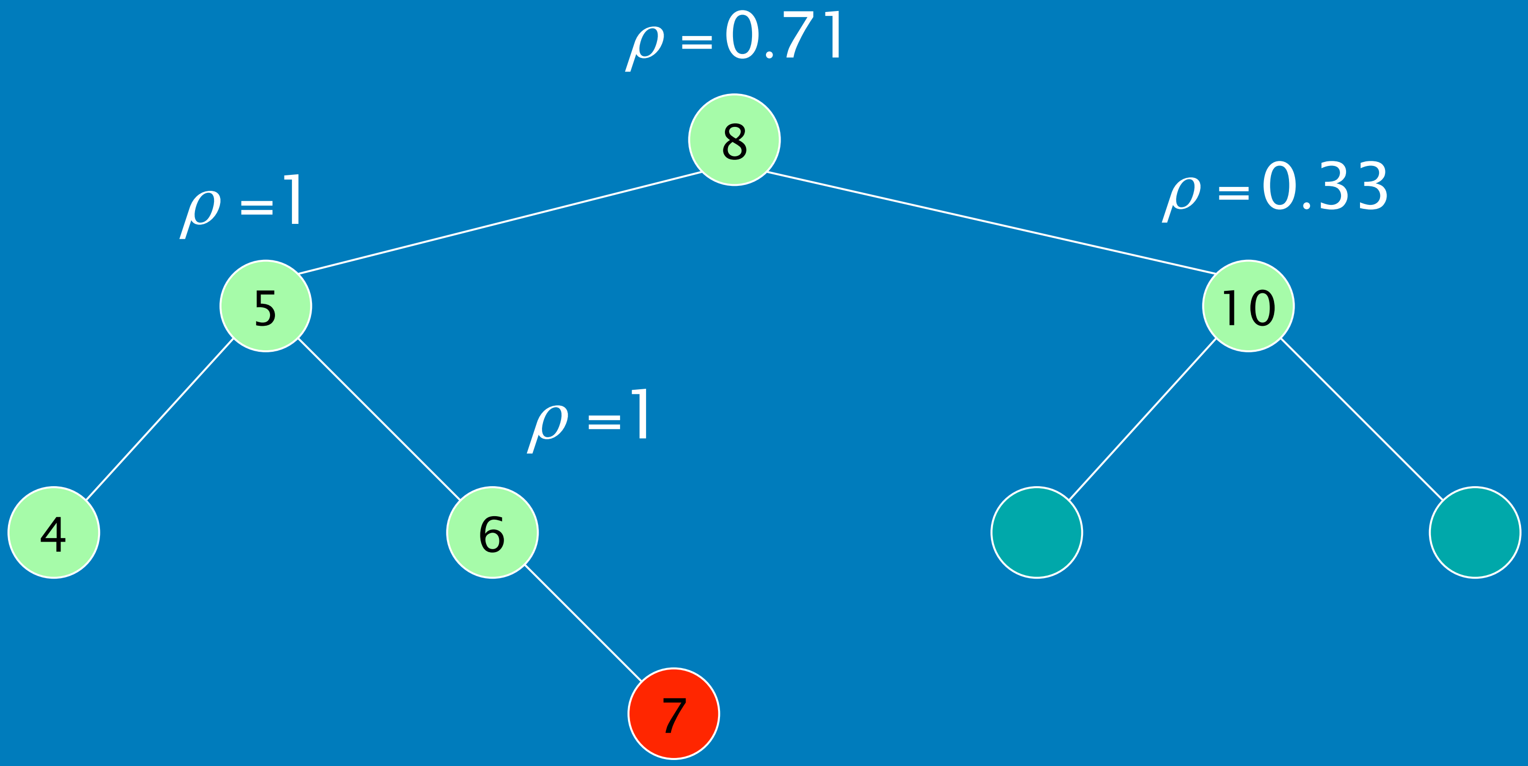
Rebalancing

- Example : insert 7



Rebalancing

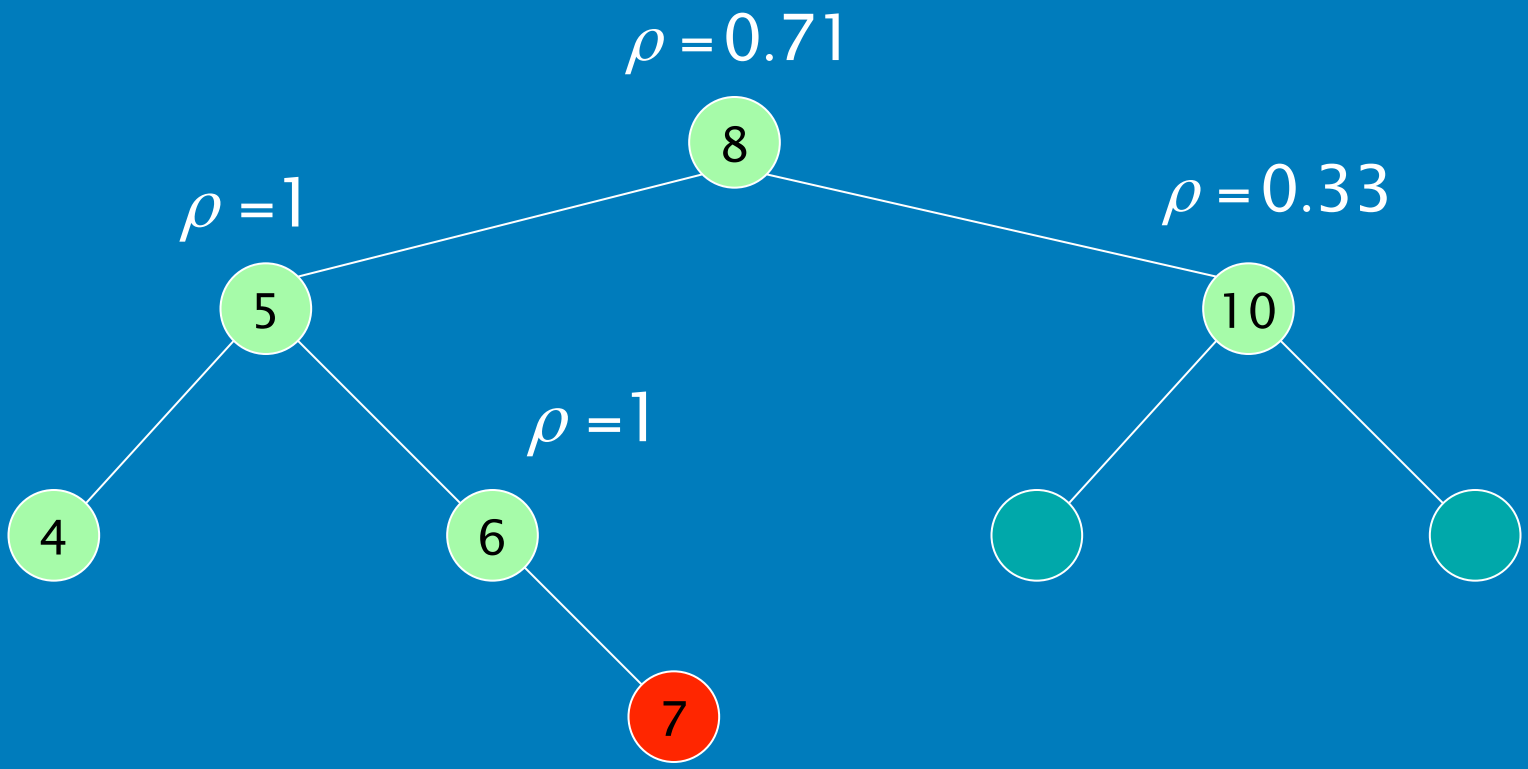
- Example : insert 7



Rebalancing

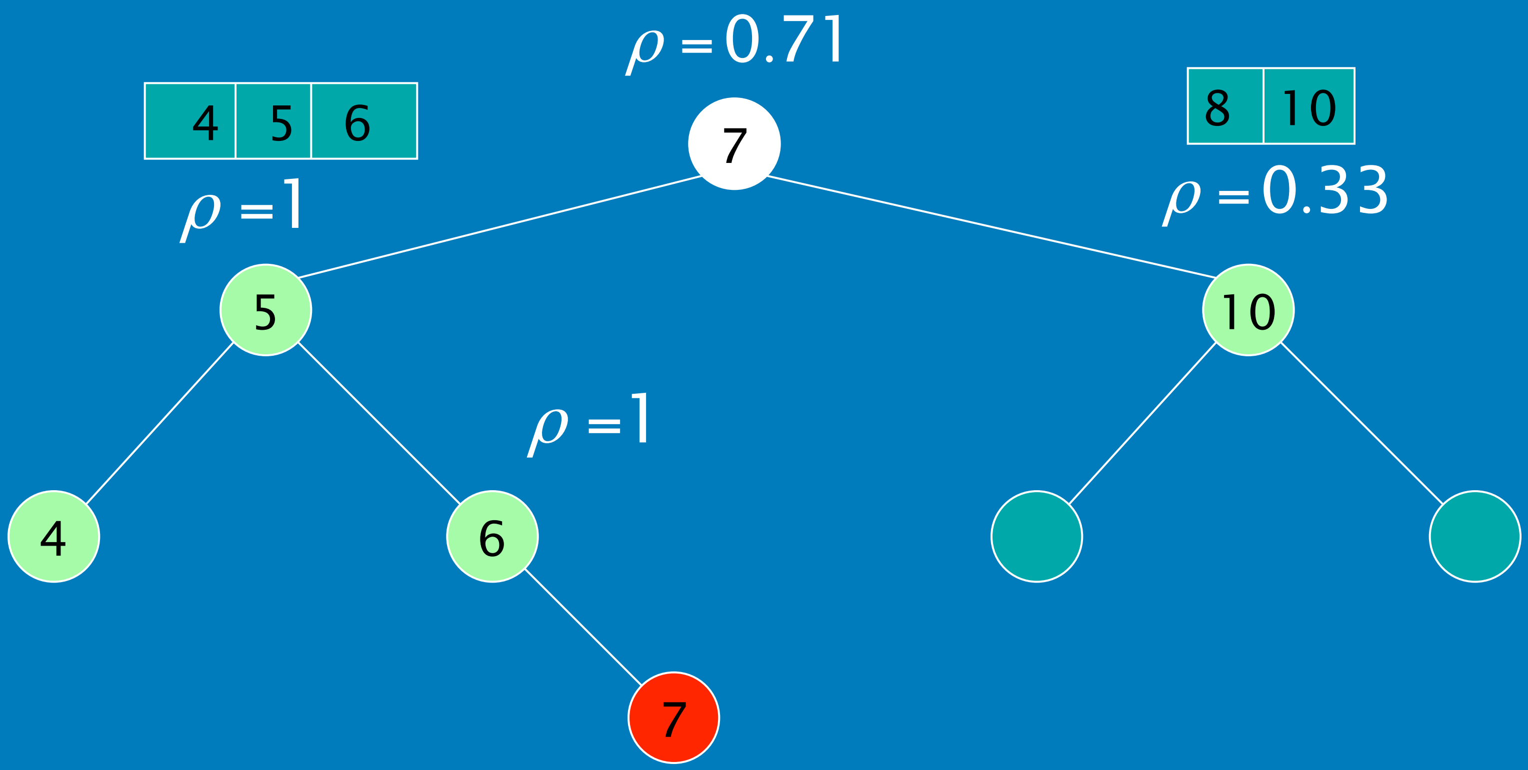
➤ Example : insert 7

4	5	6	7	8	10



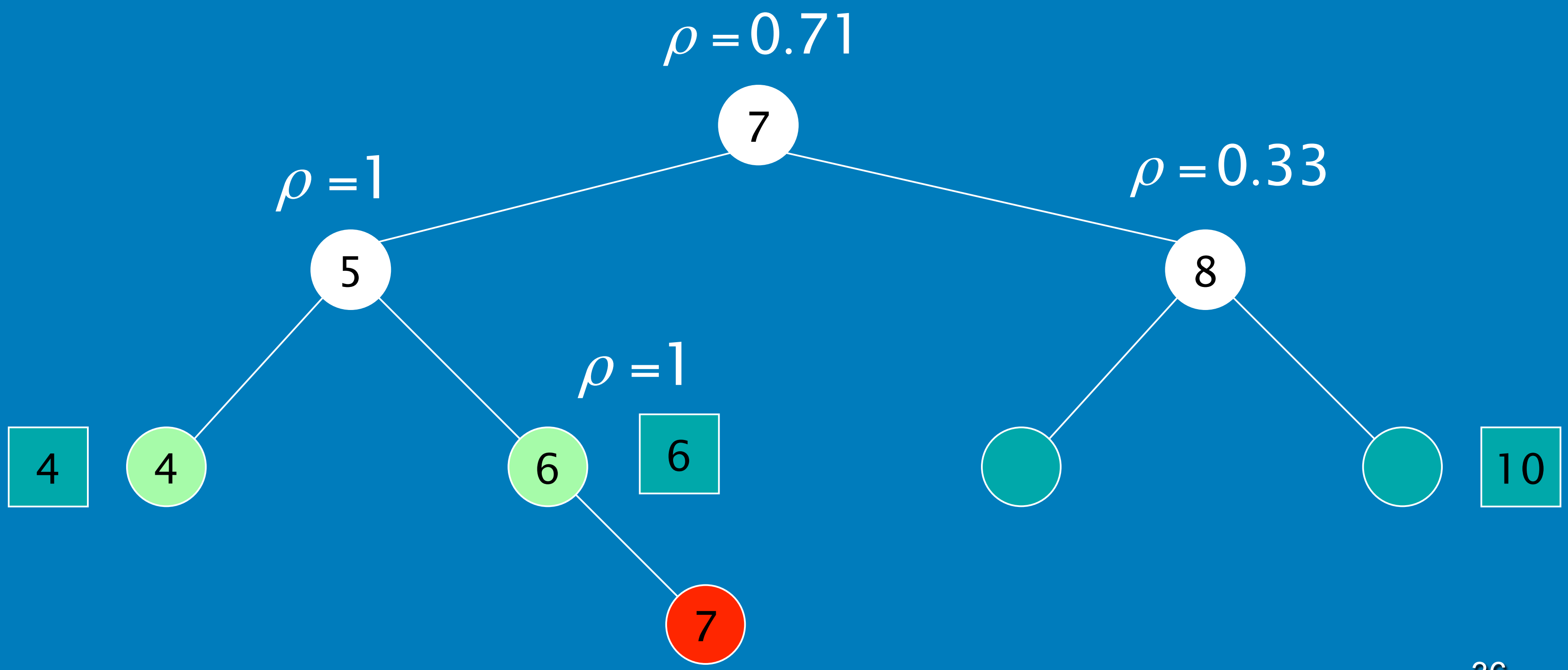
Rebalancing

- Example : insert 7



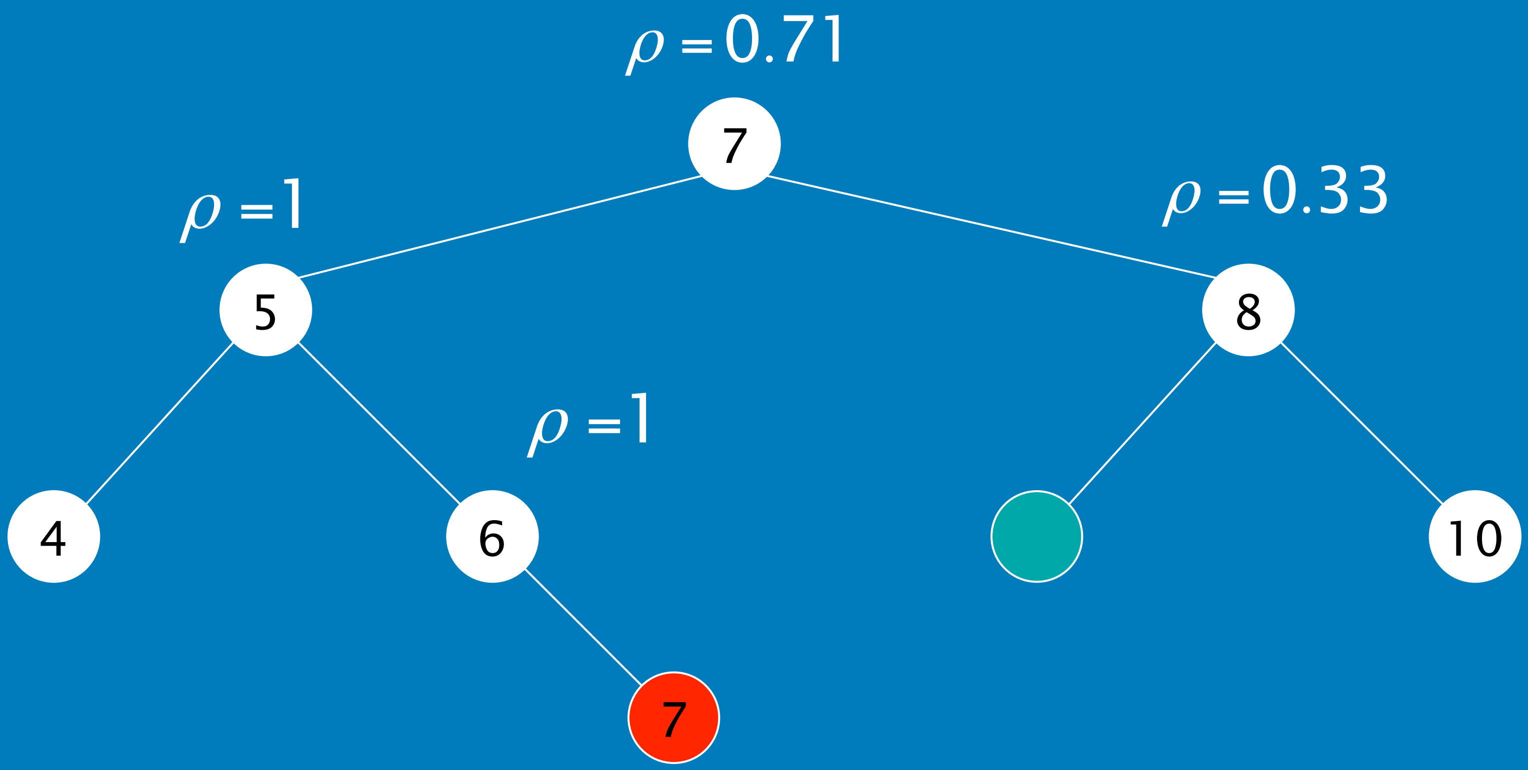
Rebalancing

- Example : insert 7



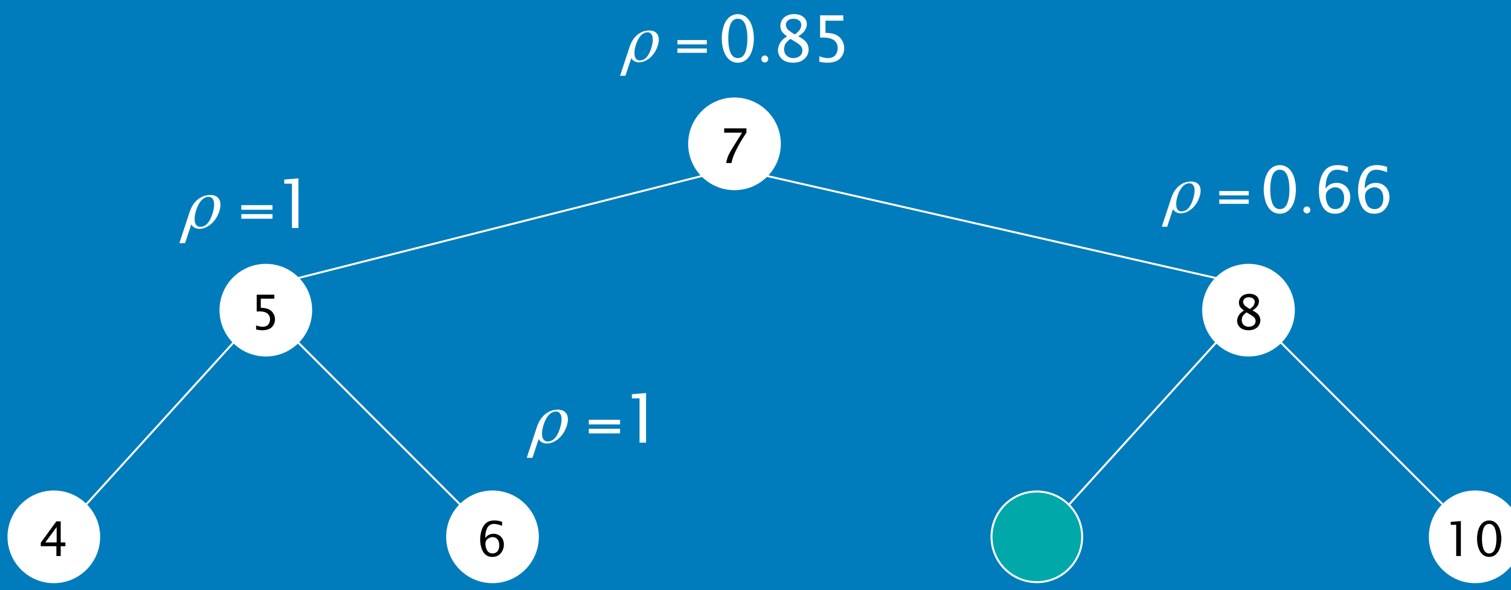
Rebalancing

- Example : insert 7



Rebalancing

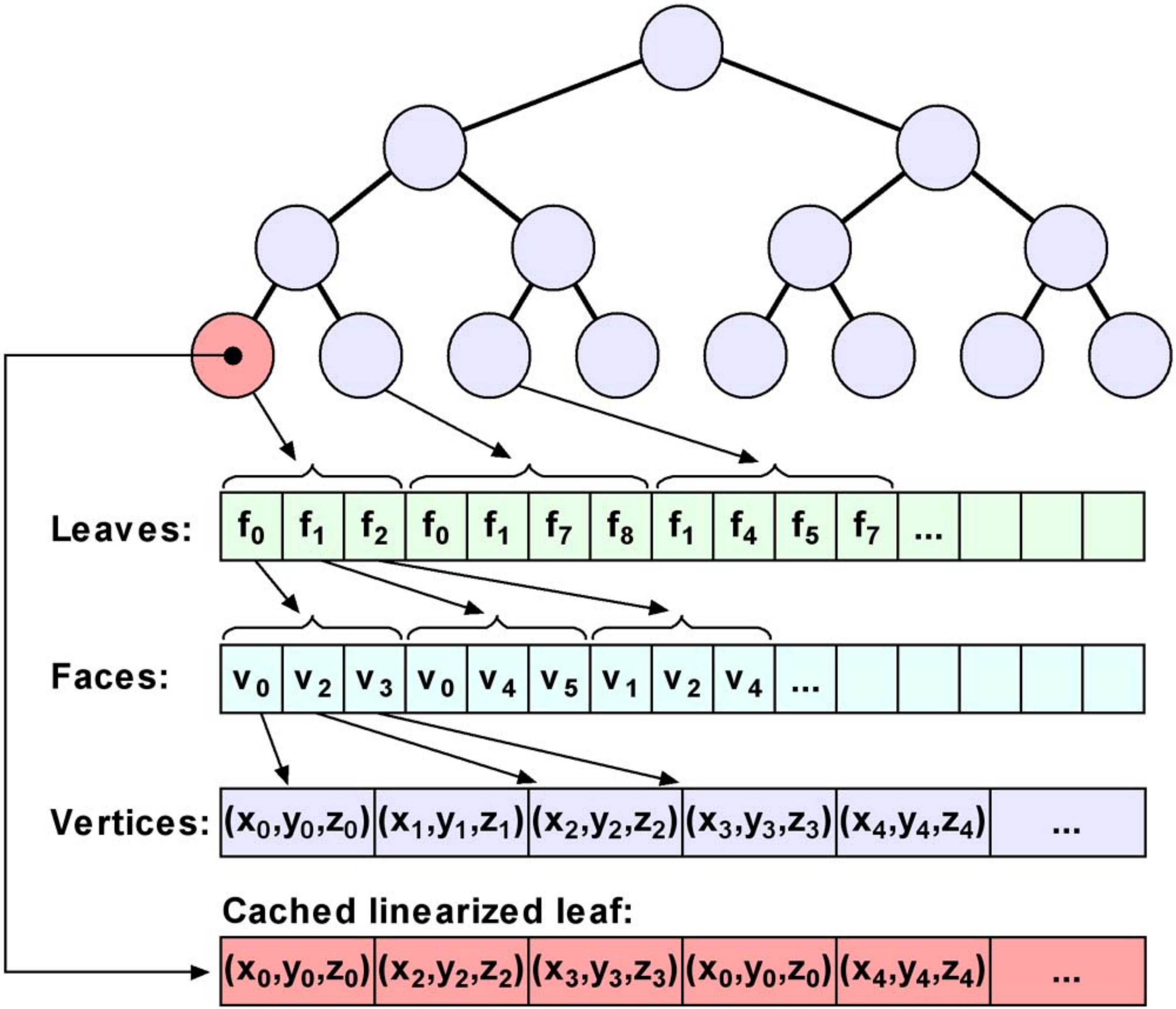
- Example : insert 7



- The next insertion will cause a rebuilding

Linearization caching

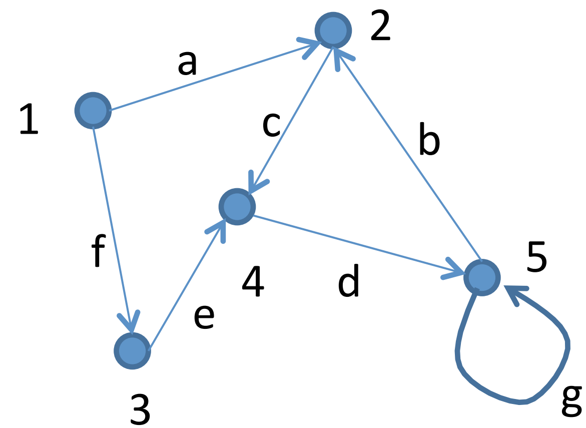
- **Nothing better than linear data**
  - Best possible spatial locality
  - Easily prefetchable
- **So linearize data at runtime!**
  - Fetch data, store linearized in a custom cache
  - Use it to linearize…
    - hierarchy traversals
    - indexed data
    - other random-access stuff



Leaves:
f0 f1 f2 f0 f1 f7 f8 f1 f4 f5 f7 ...
Faces:
v0 v2 v3 v0 v4 v5 v1 v2 v4 ...
Vertices:
(x0,y0,z0) (x1,y1,z1) (x2,y2,z2) (x3,y3,z3) (x4,y4,z4) ...
Cached linearized leaf:
(x0,y0,z0) (x2,y2,z2) (x3,y3,z3) (x0,y0,z0) (x4,y4,z4) ...

Relating graphs and matrices

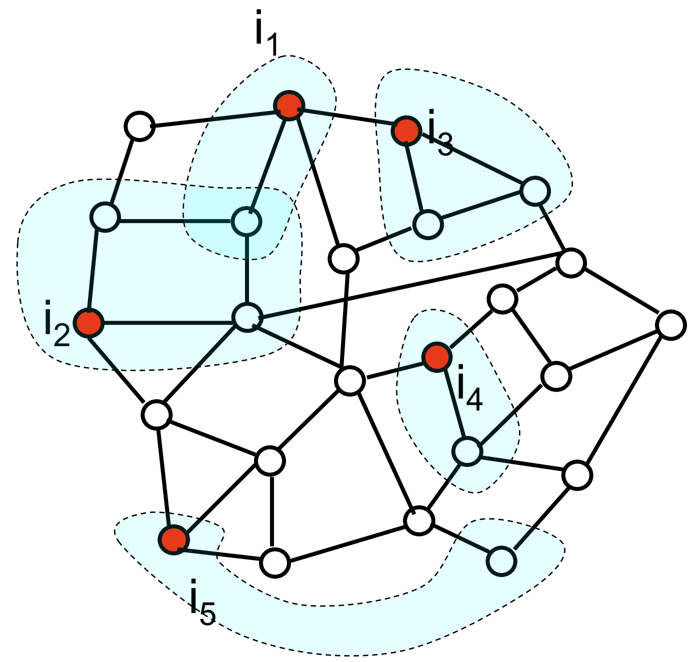
- Graphs can be viewed as matrices and vice versa
- Order of edge visits in algorithm = order of matrix entry visits
  - Row-wise traversal of matrix = visit each node of graph and walk over its outgoing edges
  - Column-wise traversal of matrix = visit each node of graph and walk over its incoming edges
  - Block traversal of matrix = ?



	1	2	3	4	5
1	0	a	f	0	0
2	0	0	0	c	0
3	0	0	0	e	0
4	0	0	0	0	d
5	0	b	0	0	g

Locality in ADP model

- Temporal locality:
  - Activities with overlapping neighborhoods should be scheduled close together in time on same core
  - Example: activities i_1 and i_2
- Spatial locality:
  - Abstract view of graph can be misleading
  - Depends on the concrete representation of the data structure
- Inter-package locality:
  - Partition graph between packages and partition concrete data structure correspondingly (see next time)
  - Active node is processed by package that owns that node



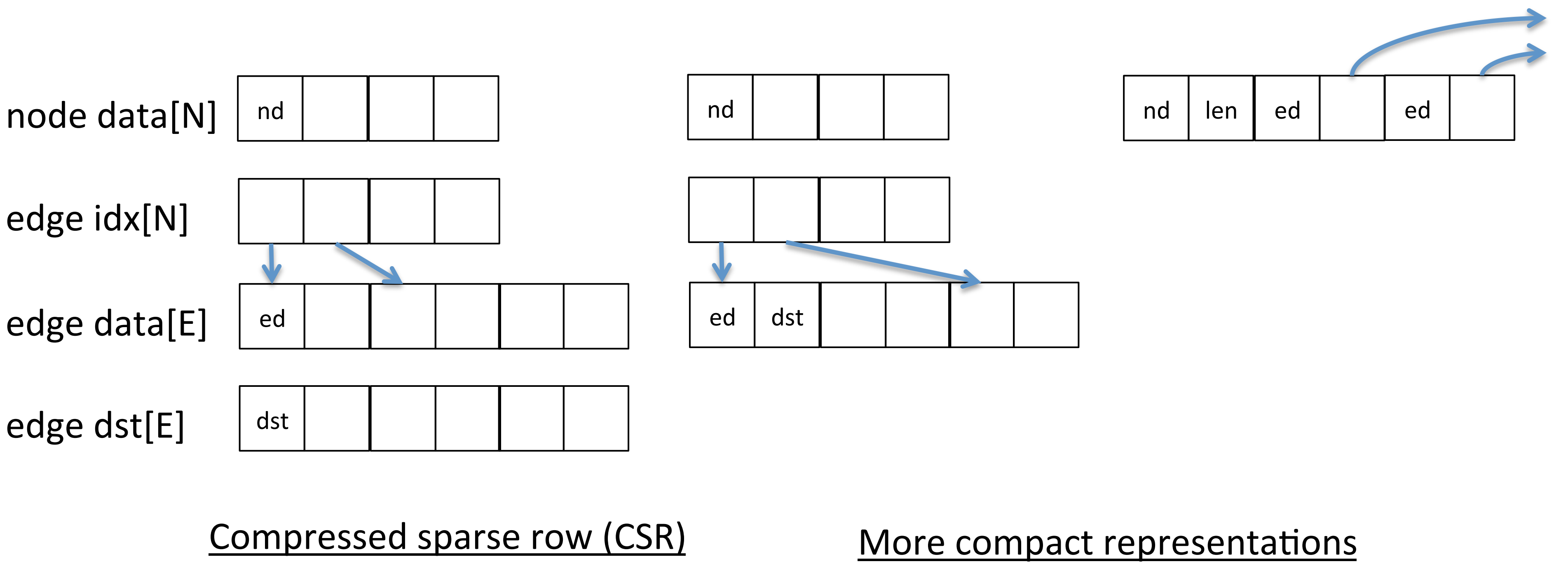
Abstract data structure

src	1	1	2	3
dst	2	1	3	2
val	3.4	3.6	0.9	2.1

Concrete representation: coordinate storage

Galois Graph

- Local computation graph:
  - Compressed sparse row (CSR) storage permits exploitation of temporal and spatial locality for algorithms that iterate over edges of a given node
  - More compact versions that inline some of the arrays in CSR format are also available



Summary
Friends: The 3 R's

- **Rearrange (code, data)**
  - Change layout to increase spatial locality
- **Reduce (size, # cache lines read)**
  - Smaller/smarter formats, compression
- **Reuse (cache lines)**
  - Increase temporal (and spatial) locality

	Compulsory	Capacity	Conflict
Rearrange	**X**	**(x)**	**X**
Reduce	**X**	**X**	**(x)**
Reuse	**(x)**	**X**	