Part II

A peek at Clojure's
Persistent Data Structures

Persistent Data Structures

- Clojure Persistent Data Structures (PDSs)
  - are immutable
  - efficient operations that take as input a PDS, and produce as output a ”similar” PDS
    - e.g., ”assoc(K, V)”: if P is a persistent hash map, create a persistent hash map Q which has the same entries as P, and additionally maps key K to value V.
    - The input and output structures share most of their data structure (which is efficient and safe)
    - The input is still available after the operation

clojure.lang.PersistentHashMap

- Implements the classical ”hash map” data structure
  - Fast hashed access,
    - `IMapEntry entryAt(Object key)`
  - Fast put operation,
    - `IPersistentMap assoc(Object key, Object val)`
- Implemented with a wide tree to share structure
  - Operations are fast constant-time in practice

Structure

- Rich Hickey created a persistent version of Phil Bagwell's ”Array-mapped hash trie.”
- Overall structure is a wide tree where there are 5 kinds of nodes, implementing interface INode:

```
static interface INode{
    INode assoc(int shift, int hash, Object key, Object val, Box addedLeaf);
    INode without(int hash, Object key);
    LeafNode find(int hash, Object key);  //and more...
}
```

  - `EmptyNode`, `LeafNode`, `FullNode`
  - `HashCollisionNode`
  - `BitmapIndexedNode` (<= this is most important)

Node ”lifecycle:” assoc (put)

- The root node of the tree is initially an `EmptyNode`.
- `LeafNodes` hold elements stored in map
- Here we do not consider `HashCollisionNodes`, or `FullNodes`
  - Special cases; go read the source ;-)
- An `EmptyNode` produces a `LeafNode` with `assoc`
- A `LeafNode` typically becomes a `BitmapIndexedNode`

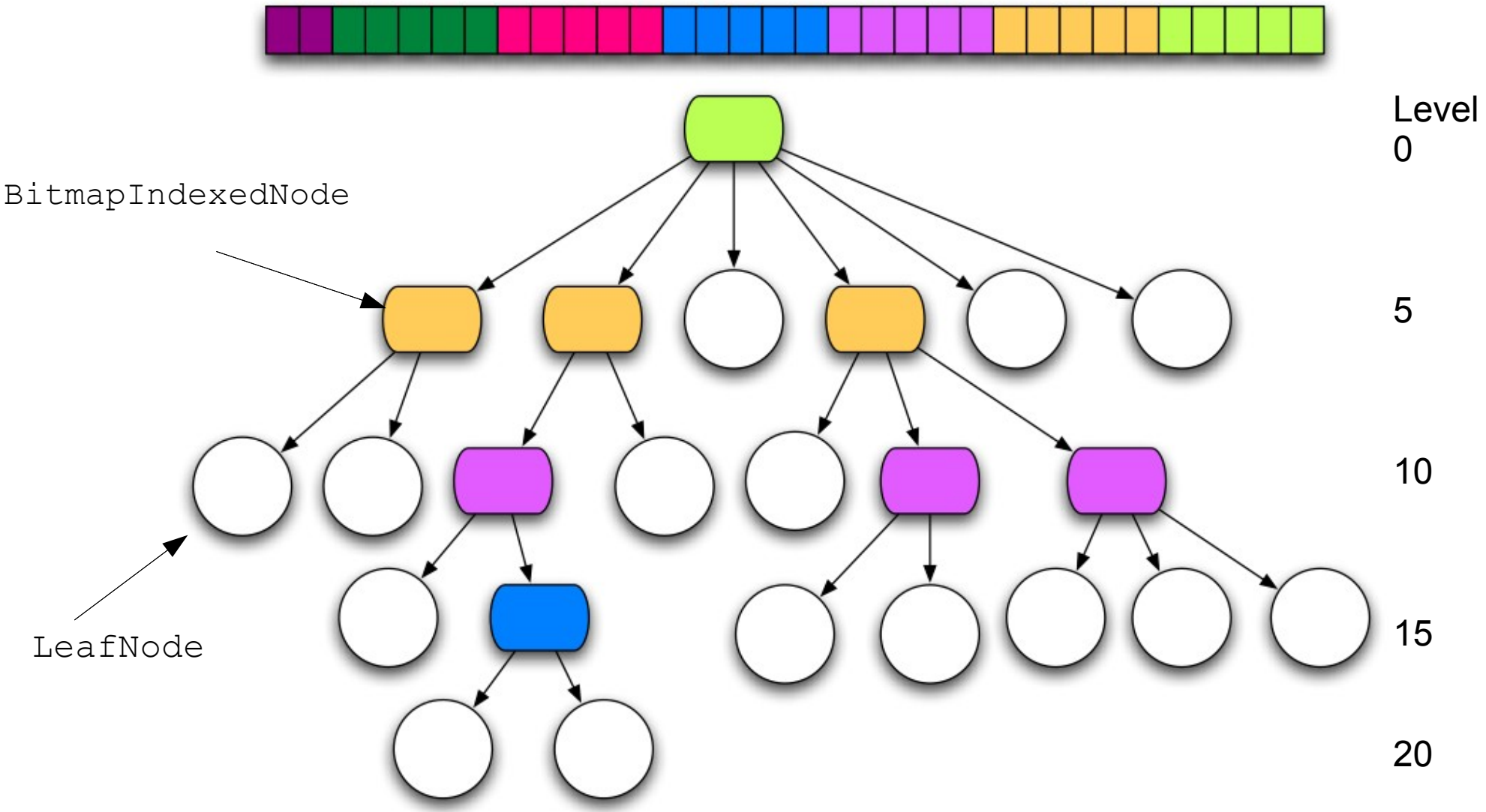
Bit partitioning

- Use partitioning of Java bit-representation of hash code.
  - Partition in blocks of 5.
  - Each block corresponds to a level in the tree
- A block is also number in [0,31]
- Exampes

```
1:      [00][00000][00000][00000][00000][00000][00001]
234:    [00][00000][00000][00000][00000][00111][01010]
1258:   [00][00000][00000][00000][00001][00111][01010]
```

Bit-partitioned Hash Trie

(slide by Rich Hickey)



BitmapIndexedNode

- Holds an array of size < 32, pointing to children
- Hard-part is to only use as much space as is needed:
  - If node has `n` children, *only* use size `n` array;
- and, doing a `lookup` on a `BitmapIndexedNode` to find a child must be fast constant time
- The trick is to find an *efficiently computable* function to map between a 5-bit number (i.e., a bit block) and index, $0 \leq i < n$ in child array

BitmapIndexedNode: The bitmap

- Consider the mapping
  - `bitpos: [0, 31] => {10^n | n ≥ 0}` (binary rep).
    - `bitpos(n) = 10^n`
  - e.g., 13 which is bit-pattern `01101` becomes
    `bitpos(011001) = 00000010000000000000`
- A bitmap is maintained which is a bit-pattern
  - e.g., `00000100000011000100010000000001`
  - so that if `i`'th bit is a 1 then there is a child with bitpos `10^i`

Bitmap: Index

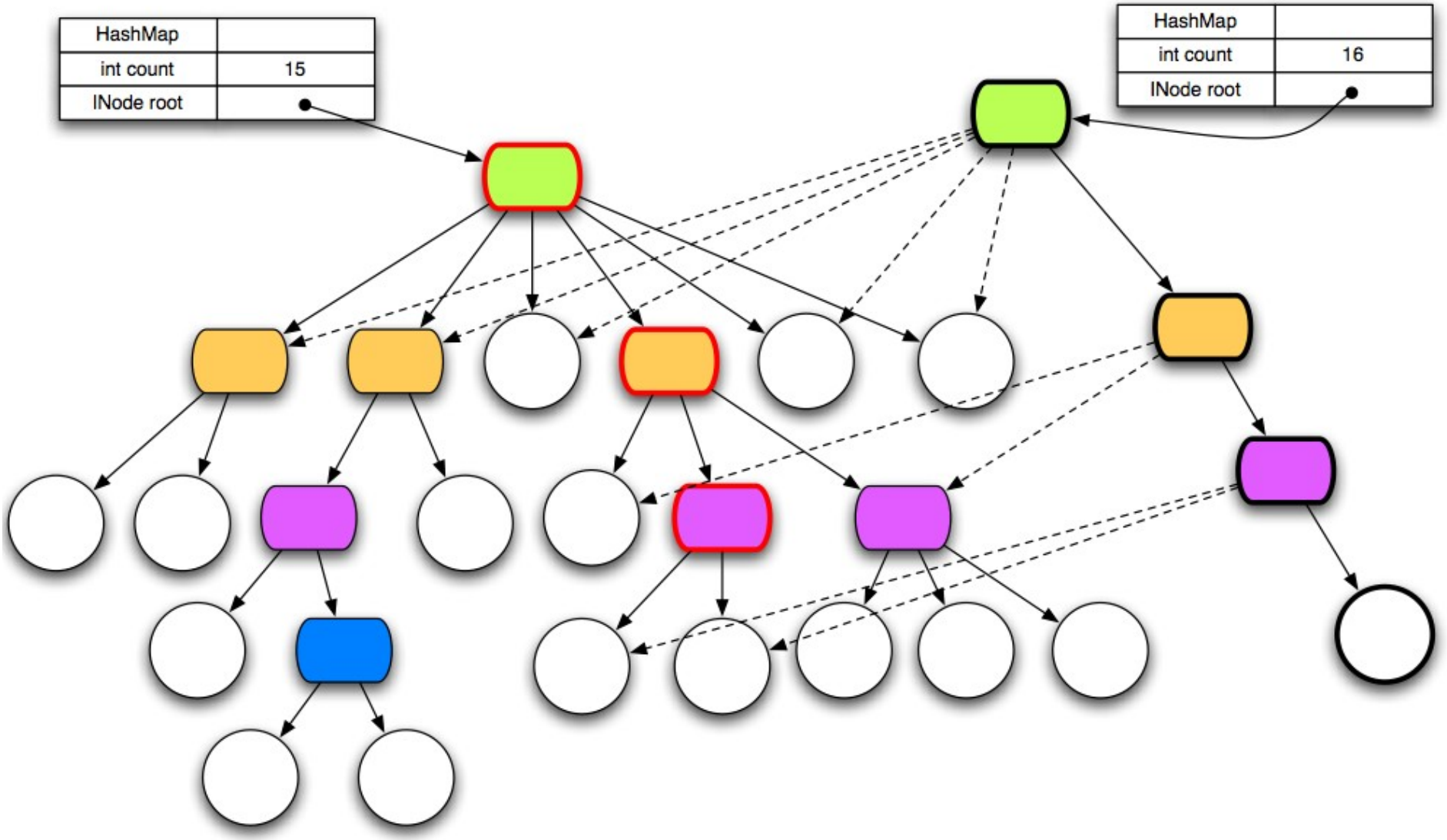
- For a given bitmap, e.g.,
  - `00000100000001100100001000000001`
- The index of an element, say with bitpos:
  - `00000000000001000000000000000000`
- Is the number of `1`'s below this bitpos, in the bitmap,
  - in the above example: 4.
- On many modern processors there is an instruction CTPOP (count population)

In code

```
static int mask(int hash, int shift){
    return (hash >>> shift) & 0x01f;
}
static int bitpos(int hash, int shift){
        return 1 << mask(hash, shift);
}
final int index(int bit){
    return Integer.bitCount(bitmap & (bit - 1));
}

public LeafNode find(int hash, Object key){
        int bit = bitpos(hash, shift);
        if((bitmap & bit) != 0)
            {
            return nodes[index(bit)].find(hash, key);
            }
        else
            return null;
}
```

Path Copying (for PersistentHashMap)
(slide by Rich Hickey)



References

- The code ;-)
- My blog for a longer description
  - http://blog.higher-order.net
- PersistentHashMap
- http://blog.higher-order.net/2009/09/08/understanding-clojures-persistenthashmap-deftwice/
- PersistentVector
  - http://blog.higher-order.net/2009/02/01/understanding-clojures-persistentvector-implementation/