Fusion Trees

Part Two

Recap from Last Time

Ordered Dictionaries

- An ***ordered dictionary*** maintains a set S drawn from an ordered universe $\mathscr{U}$ and supports these operations:
  - ***lookup***(x), which returns whether $x \in S$;
  - ***insert***(x), which adds x to S;
  - ***delete***(x), which removes x from S;
  - ***max***() / ***min***(), which return the maximum or minimum element of S;
  - ***successor***(x), which returns the smallest element of S greater than x; and
  - ***predecessor***(x), which returns the largest element of S smaller than x.

Ordered Dictionary : BST :: Queue : Linked List

Our Machine Model

- We will assume we're working on a machine where memory is segmented into ***w***-bit words.
- We'll assume that the C integer operators work in constant time, and will not assume we have access to operators beyond them.

```
+   -   *   /   %   <<   >>   &   |   ^   ==   <=
```

Word-Level Parallelism

- Last time, we saw five powerful primitives built using word-level parallelism:
  - ***Parallel compare:*** We can compare a bunch of small numbers in parallel in O(1) machine word operations.
  - ***Parallel tile:*** We can take a small number and “tile” it multiple times in O(1) machine word operations.
  - ***Parallel add:*** If we have a bunch of “flag” bits spread out evenly, we can add them all up in O(1) machine word operations.
  - ***Parallel rank:*** We can find the rank of a small number in an array of small numbers in O(1) machine word operations.
  - ***Most-significant bit:*** We can compute `msb`(n) for any w-bit integer n in O(1) machine word operations.

Word-Level Parallelism

- Last time, we saw five powerful primitives built using word-level parallelism:
  - ***Parallel compare:*** We can compare a bunch of small numbers in parallel in O(1) machine word operations.
  - ***Parallel tile:*** We can take a small number and "tile" it multiple times in O(1) machine word operations.
  - ***Parallel add:*** If we have a bunch of "flag" bits spread out evenly, we can add them all up in O(1) machine word operations.
  - ***Parallel rank:*** We can find the rank of a small number in an array of small numbers in O(1) machine word operations.
  - ***Most-significant bit:*** We can compute `msb`(n) for any w-bit integer n in O(1) machine word operations.

Integer LCP

- Computing `msb` efficiently lets us implement a number of other efficient primitives.
- Given two integers m and n, the ***longest common prefix*** of m and n, denoted **`lcp`(m, n)**, is the length of the longest bitstring they both start with.
- ***Claim:*** We can compute this in time O(1).

```
  00011010 01101110 01111000 01001101 00101111 00001101 01110111 01100001
⊕ 00011010 01000101 00010100 00100000 01010000 00100010 01000100 00001000
  -----------------------------------------------------------------------
  00000000 00101011 01100100 01101101 01111111 00101111 00110011 01101001
```

$$m \oplus n$$

Integer LCP

- Computing `msb` efficiently lets us implement a number of other efficient primitives.
- Given two integers m and n, the ***longest common prefix*** of m and n, denoted **`lcp`(*m*, *n*)**, is the length of the longest bitstring they both start with.
- ***Claim:*** We can compute this in time O(1).

	00011010 01101110 01111000 01001101 00101111 00001101 01110111 01100001
⊕	00011010 01000101 00010100 00100000 01010000 00100010 01000100 00001000
	00000000 00101011 01100100 01101101 01111111 00101111 00110011 01101001

$m \oplus n$

Integer LCP

- Computing `msb` efficiently lets us implement a number of other efficient primitives.
- Given two integers m and n, the ***longest common prefix*** of m and n, denoted **`lcp`(m, n)**, is the length of the longest bitstring they both start with.
- ***Claim:*** We can compute this in time O(1).

	`00011010 01101110 01111000 01001101 00101111 00001101 01110111 01100001`
⊕	`00011010 01000101 00010100 00100000 01010000 00100010 01000100 00001000`
	`00000000 00101011 01100100 01101101 01111111 00101111 00110011 01101001`

`63 - msb`$(m \oplus n)$

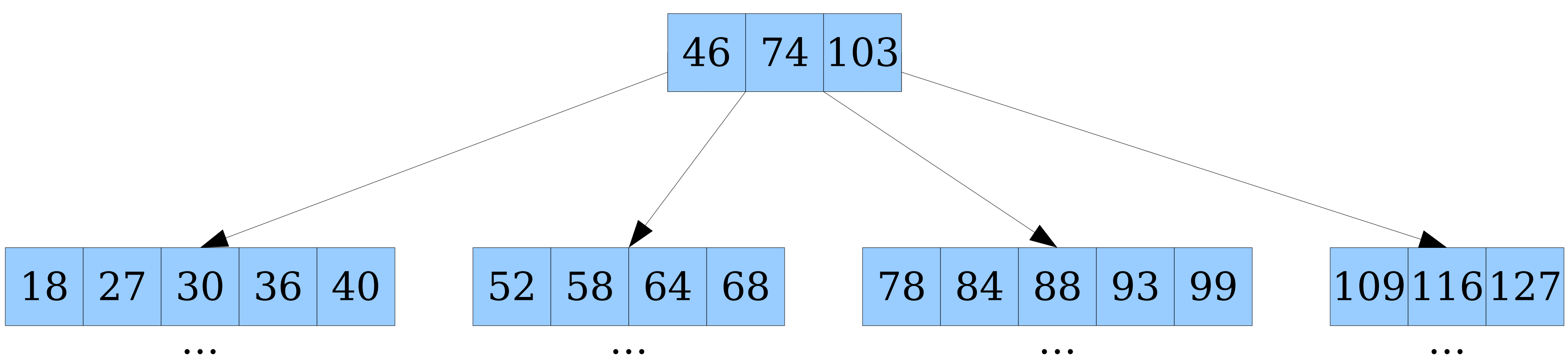
New Stuff!

The Sardine Tree Revisited

- Last time, we designed a data structure nicknamed the ***sardine tree*** that
  - stores s-bit keys, where s is much smaller than w, and
  - supports all operations in time $O(\log_{w/s} n)$.
- Our goal for today will be to generalize this to work with arbitrary integer keys, not just s-bit keys.

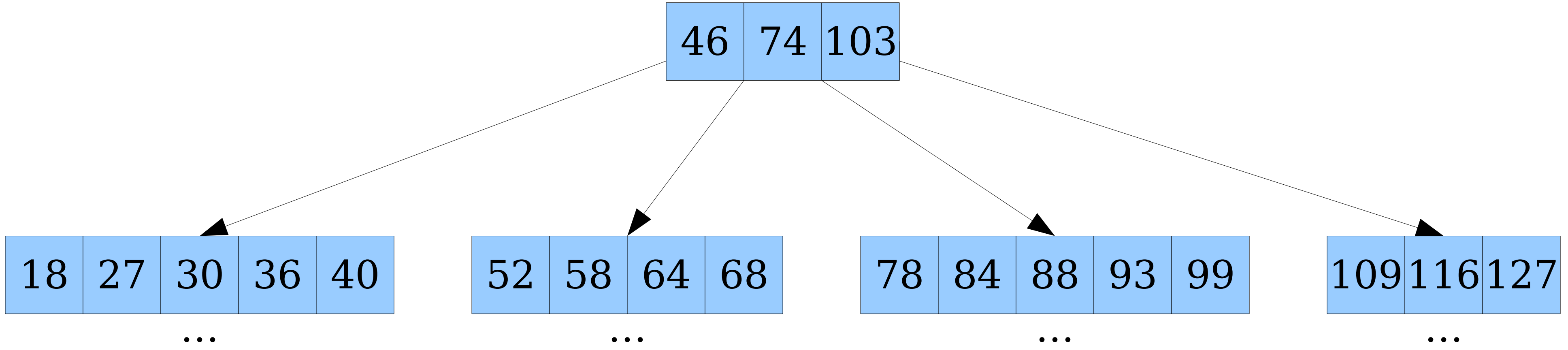
The Sardine Tree Revisited

- At a high level, the sardine tree is a B-tree augmented with extra information to support fast rank queries.
- The branching factor is $\Theta(w / s)$, the number of keys we can fit into a single machine word.
- We use a ***parallel rank*** operation at each node to determine which keys to check and which child to descend into.
- Therefore, each operation's cost is $O(\log_{w/s} n)$: $O(1)$ work per each of $O(\log_{w/s} n)$ nodes visited.



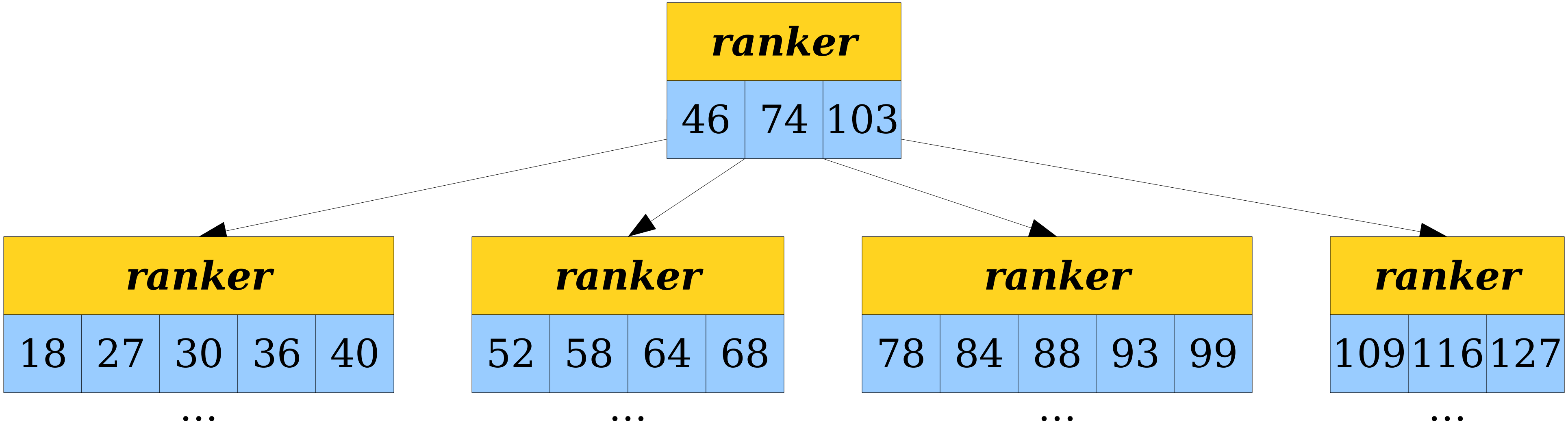
The Sardine Tree Revisited

- The sardine tree is a specific case of a more general framework.
- Build a B-tree where each node is augmented with a data structure called a ***ranker*** with the following properties:
  - The ranker stores $\Theta(K)$ total keys.
  - It supports queries of the form ***rank***(x), which returns the rank of x among those keys, in time $O(1)$.



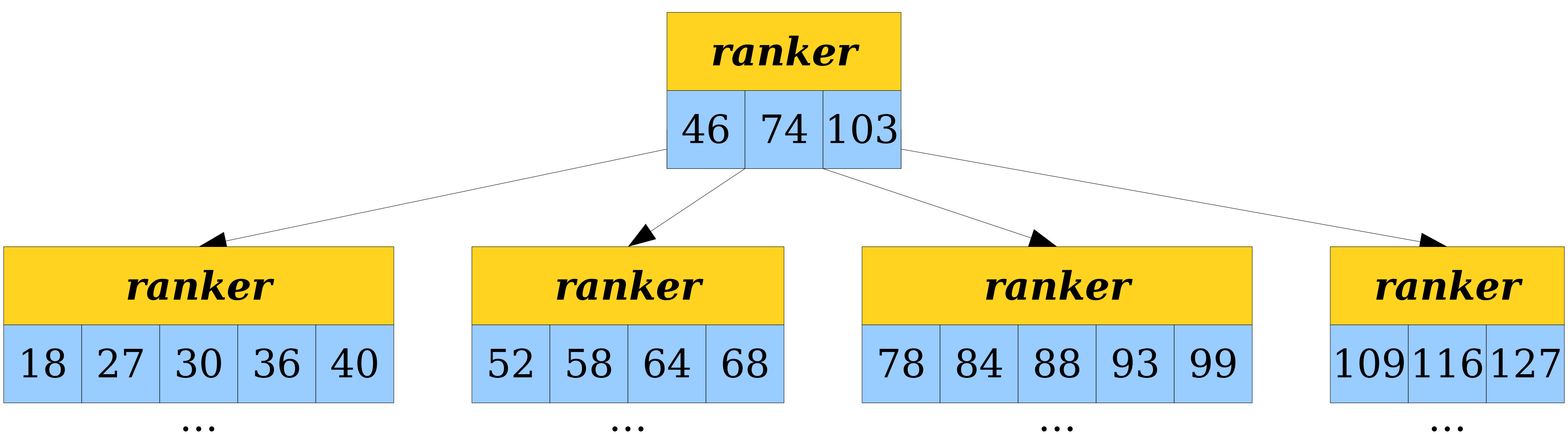
The Sardine Tree Revisited

- The sardine tree is a specific case of a more general framework.
- Build a B-tree where each node is augmented with a data structure called a ***ranker*** with the following properties:
  - The ranker stores $\Theta(K)$ total keys.
  - It supports queries of the form ***rank***(x), which returns the rank of x among those keys, in time O(1).
- The cost of performing a search is then $O(\log_K n)$, since the tree height is $O(\log_K n)$ and we do O(1) work per node.



The Sardine Tree Revisited

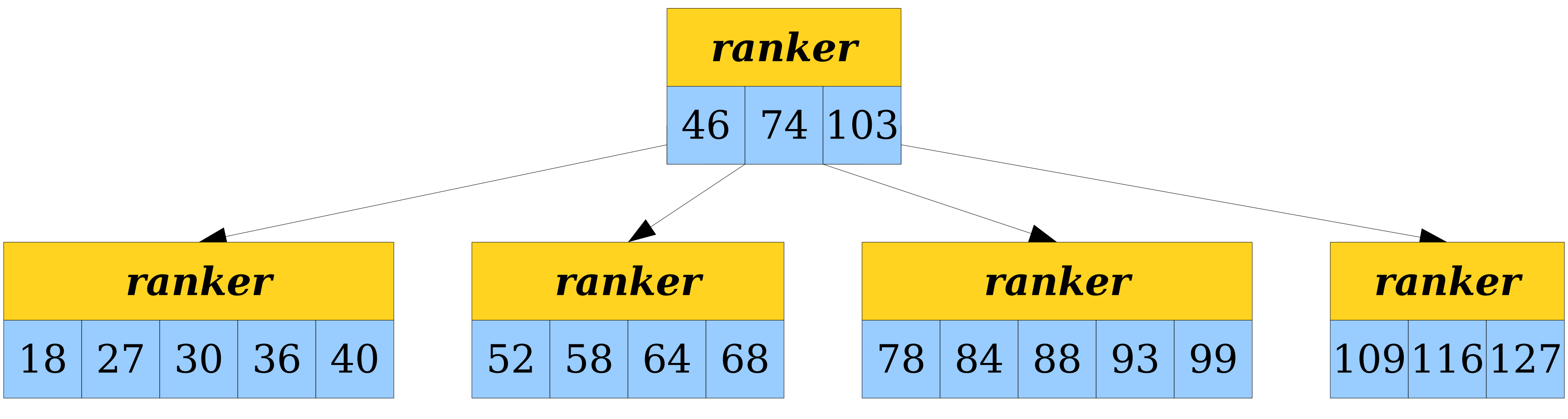
- The sardine tree ranker works by packing the $\Theta(w/s)$ keys into a machine word, then using our ***parallel rank*** operation from last time.
- Since there are $\Theta(w/s)$ keys per node, the runtime of each B-tree operation is $O(\log_{w/s} n)$, though the keys are severely size-limited.



Fusion Trees

- The ***fusion tree*** is a B-tree augmented with a ranker that stores w^{ε} keys for some constant ε. Those keys are full w-bit words.
- The cost of a lookup, successor, or predecessor in a fusion tree is therefore

$$O(\log_{w^{\varepsilon}} n) = O(\log n \mid \log w^{\varepsilon}) = \mathbf{O(\log_w n)}.$$



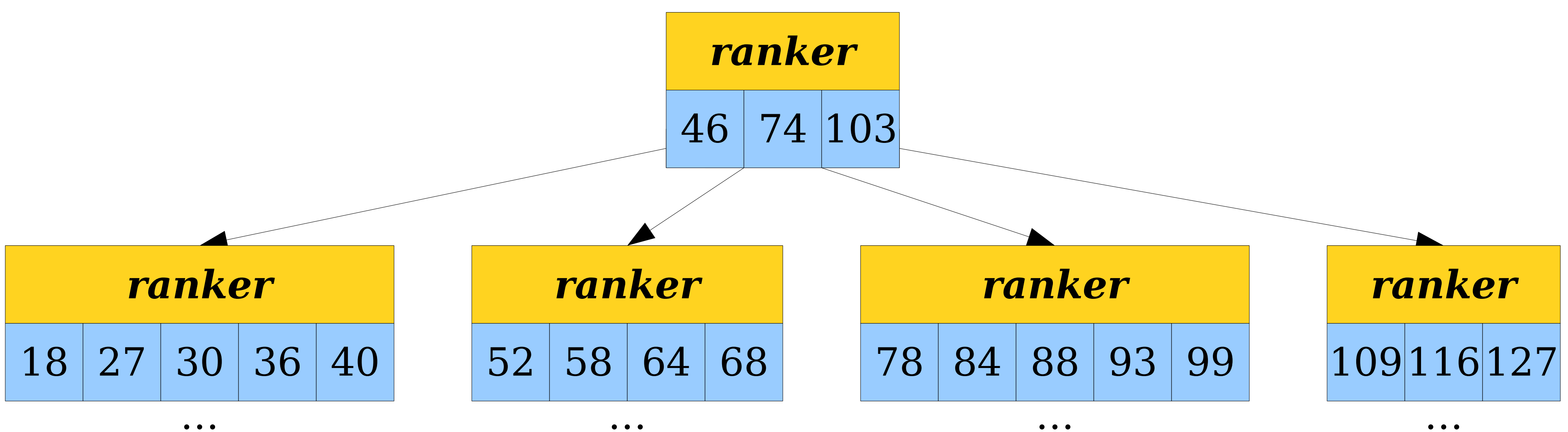
Where We're Going

- The sardine tree solves the following problem:

Support rank queries for a *large* number of *small* keys.

- To build the fusion tree, we'll solve this problem:

Support rank queries for a *small* number of *large* keys.



Where We're Going

- The ***parallel rank*** operation we devised last time permits O(1)-time rank queries, provided that all the keys fit into a machine word.
- In general, we can't assume that a collection of arbitrary keys all fit into a machine word.
- ***Goal:*** Compress multiple w-bit keys so that
  - they fit in a machine word so we can use ***parallel rank***, and
  - the compression preserves enough information about their order so that the ranks we get back are meaningful.

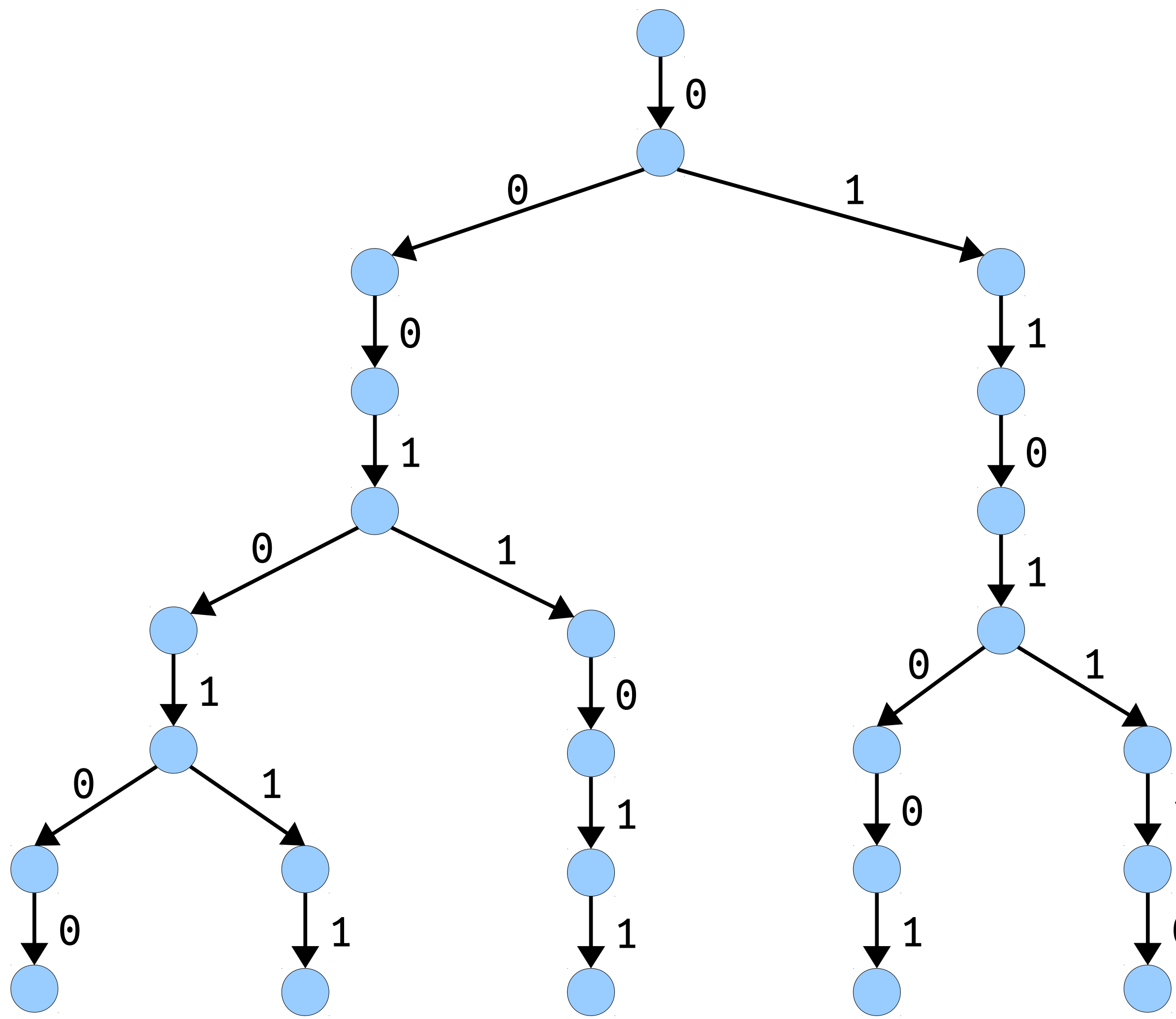
Compressing Our Numbers

- Let's imagine we have a collection of $\boldsymbol{w^{\varepsilon}}$ numbers, each of which is w bits long.
- For simplicity, we're going to assume that those numbers are given to us in advance and in sorted order.
  - We'll relax this later on.

00010100	00010111	00011011	01101001	01101110

Back to Tries

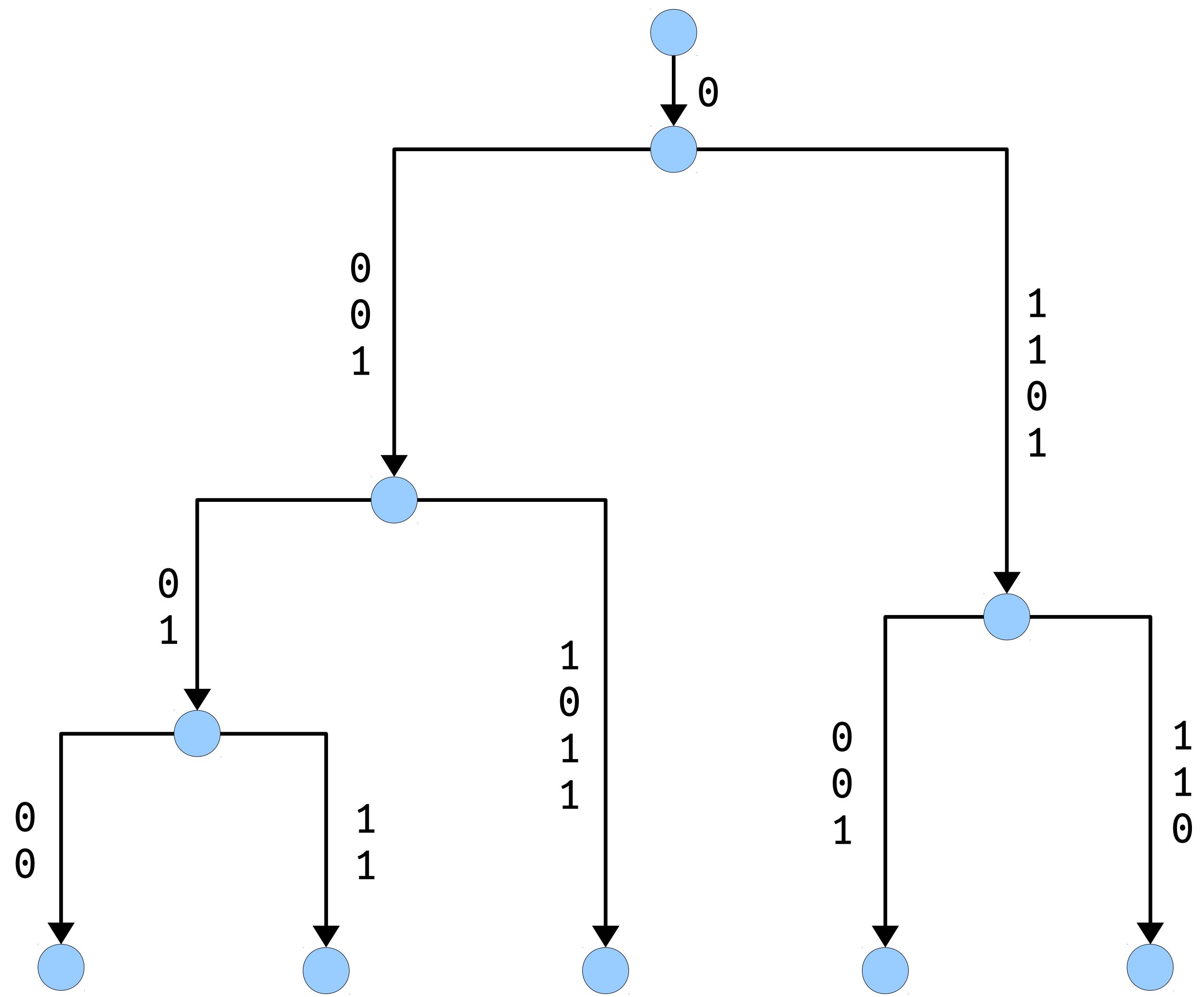
- Think about what happens if we make a trie from these numbers.
- We have few numbers (w^{ε}) and these numbers are large (size w), so most nodes will have one child.
- ***Idea:*** Use a Patricia trie!



00010100	00010111	00011011	01101001	01101110

Back to Tries

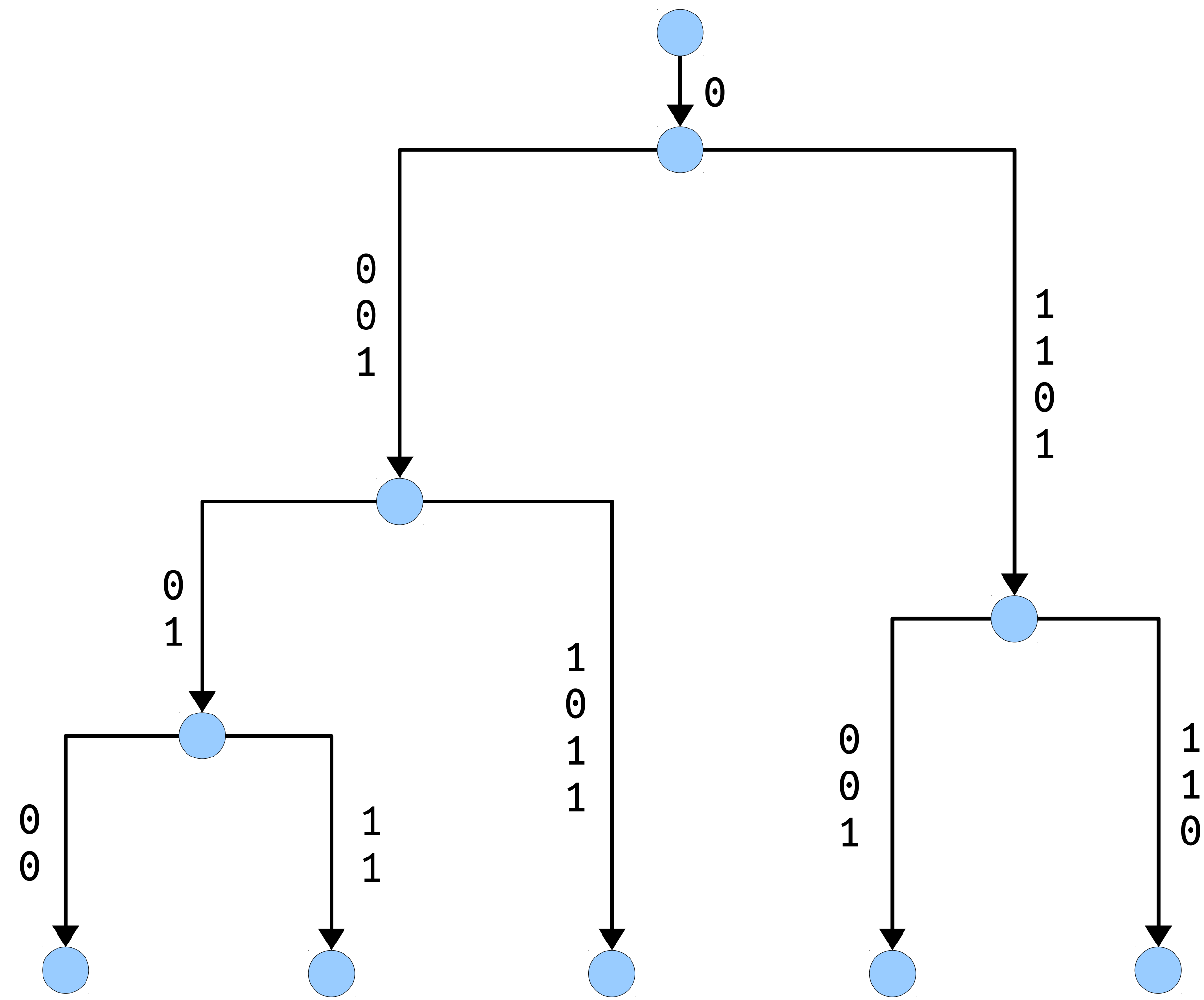
- Think about what happens if we make a trie from these numbers.
- We have few numbers (w^{ε}) and these numbers are large (size w), so most nodes will have one child.
- ***Idea:*** Use a Patricia trie!



00010100	00010111	00011011	01101001	01101110

Back to Tries

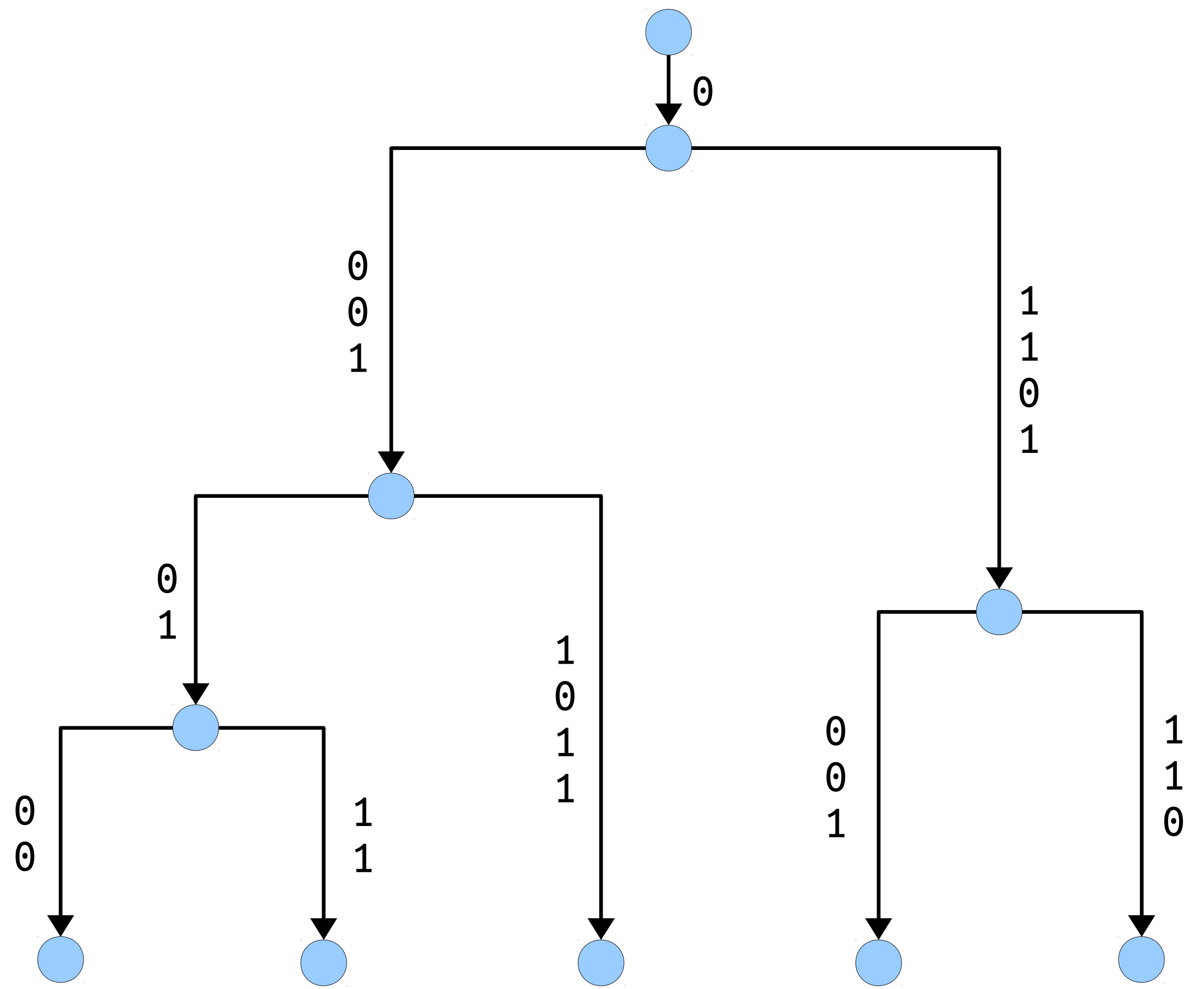
- Since there are w^ε numbers, there are exactly $w^\varepsilon - 1$ junctions in the Patricia trie.
- Look at each number and focus purely on the bits that correspond to those junctions.



00010100	00010111	00011011	01101001	01101110

Back to Tries

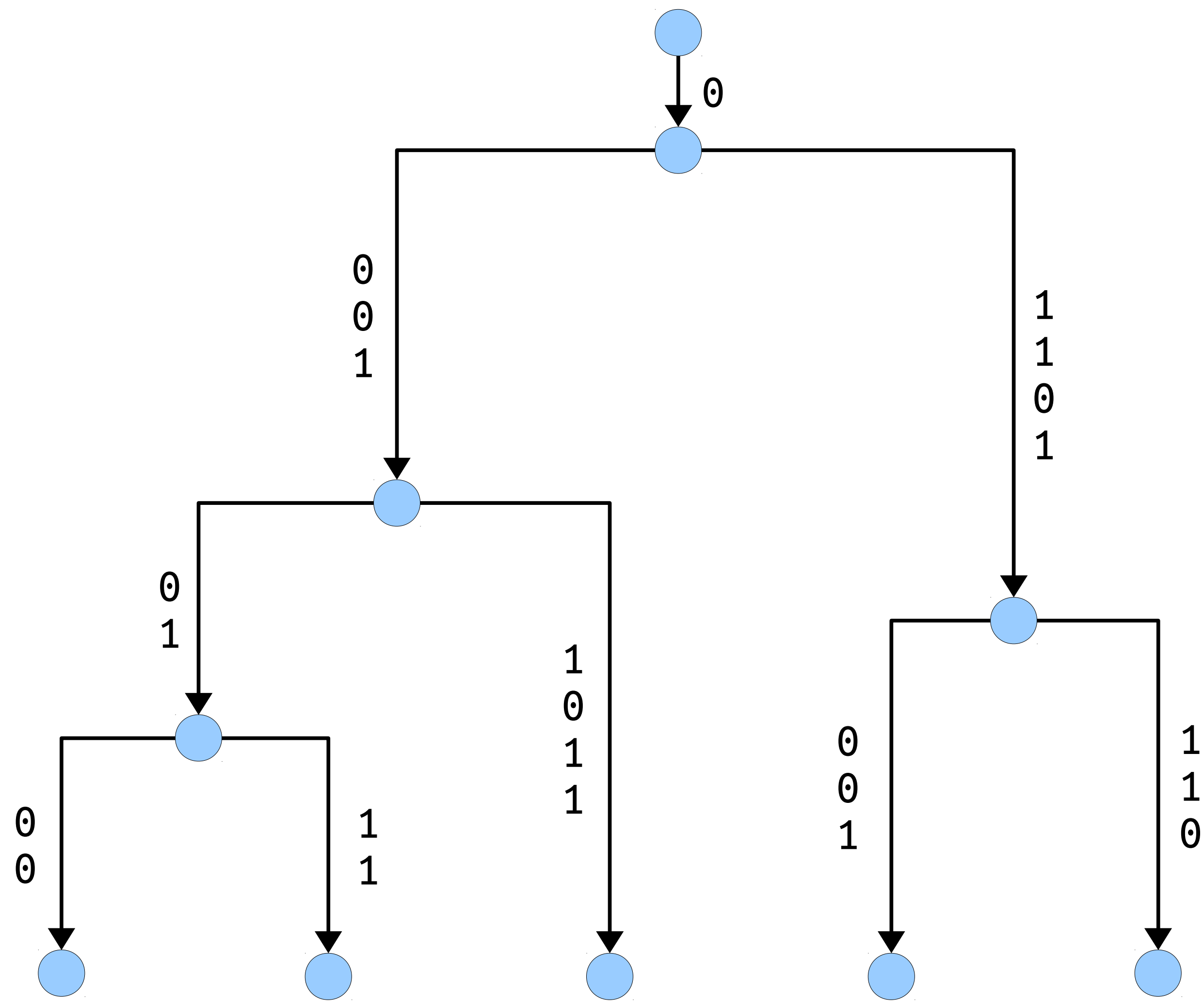
- Since there are w^{ε} numbers, there are exactly $w^{\varepsilon} - 1$ junctions in the Patricia trie.
- Look at each number and focus purely on the bits that correspond to those junctions.



000	001	01	10	11
0**0**01**0**1**0**0	0**0**01**0**1**1**1	0**0**01**1**011	0**1**101**0**01	0**1**101**1**10

Back to Tries

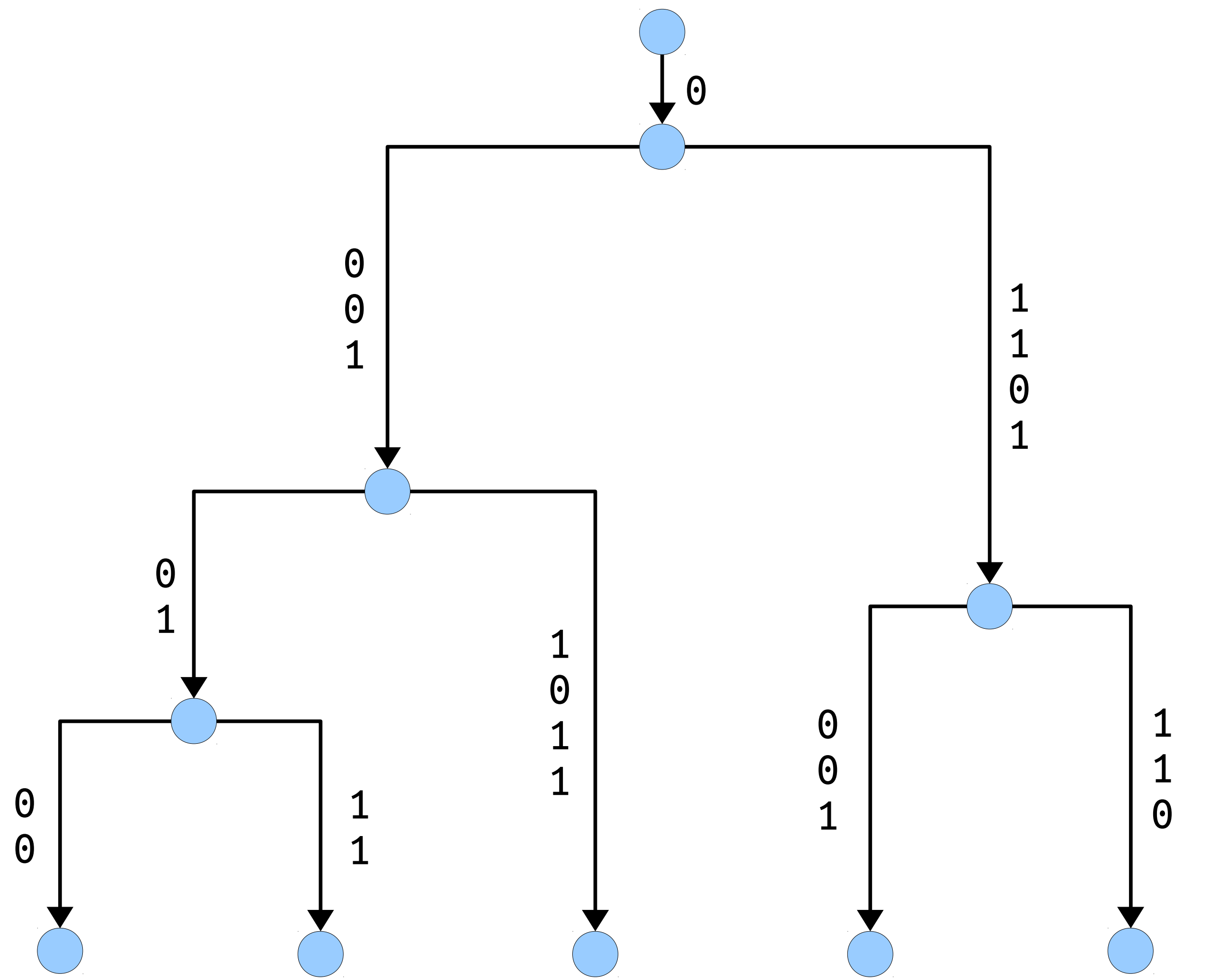
- ***Claim:*** The sorted order of these original numbers matches the *lexicographical* order of these new bitstrings.
- ***Proof idea:*** These new bitstrings represent paths through the Patricia trie.



000	001	01	10	11
00010100	00010111	00011011	01101001	01101110

Back to Tries

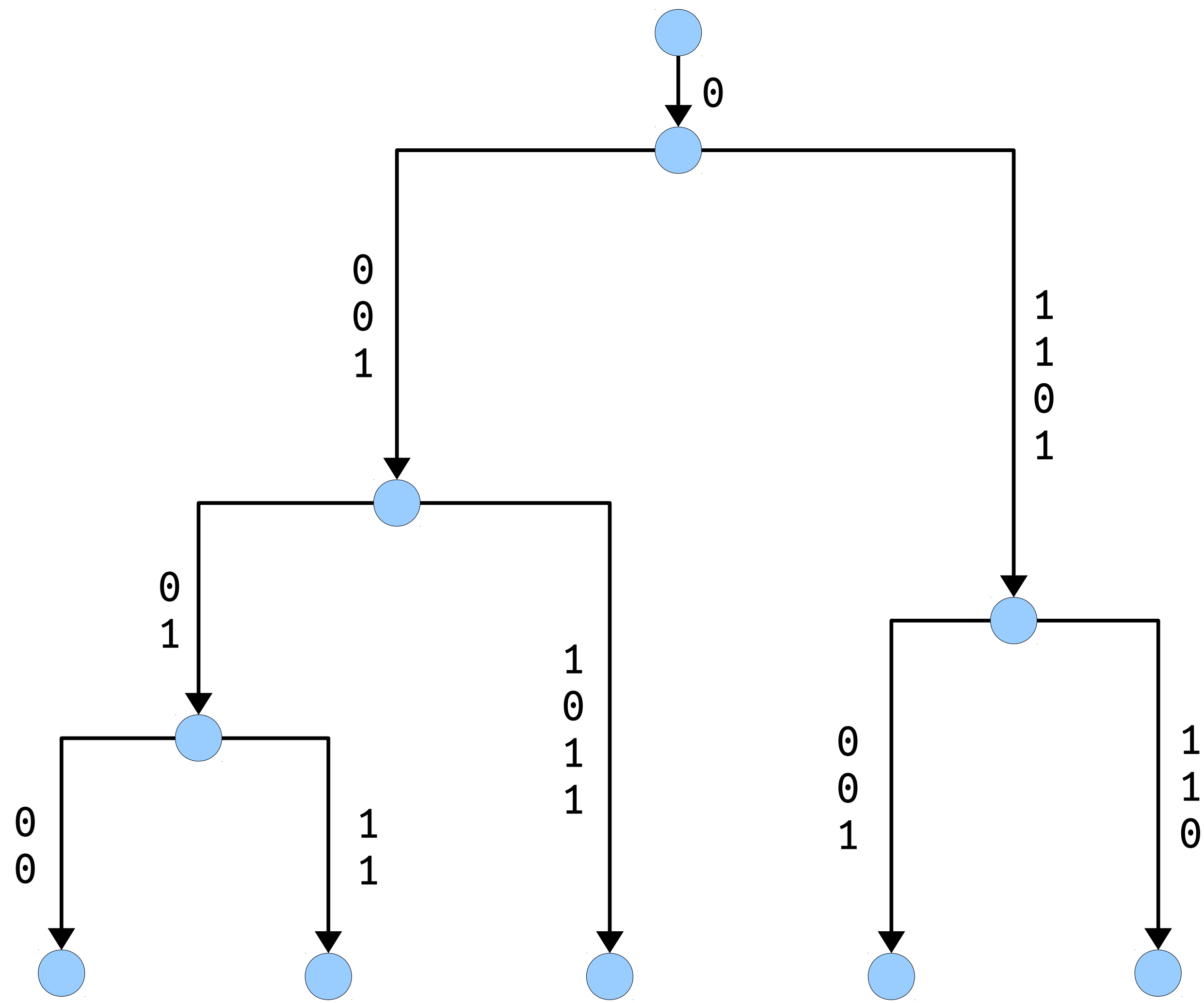
- We're ultimately interested in compressing our numbers so they all fit in a machine word.
- There are at most w^{ε} bits in each of these new numbers – that's really promising!



000	001	01	10	11
00010100	00010111	00011011	01101001	01101110

Back to Tries

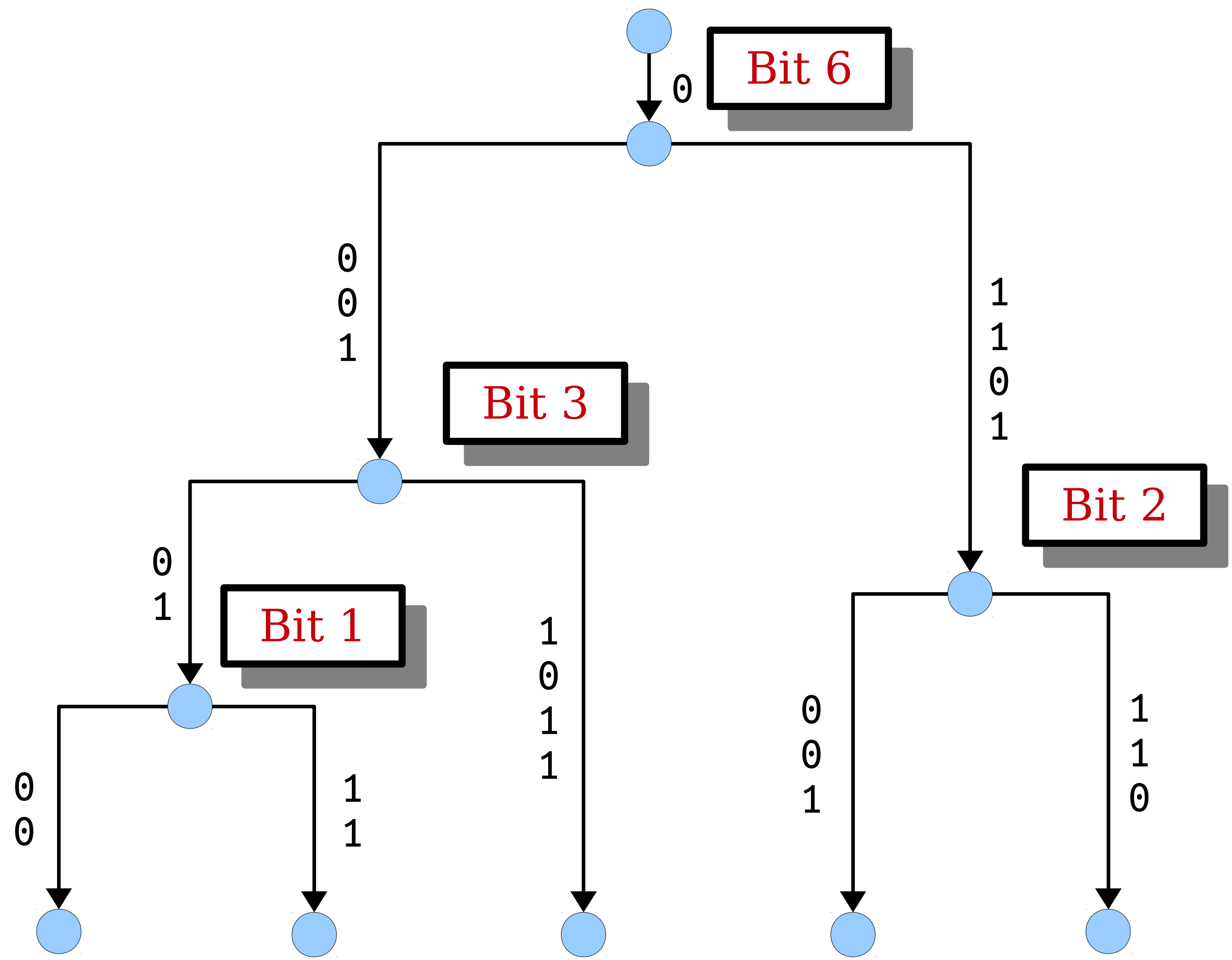
- ***Problem:*** While the *lexicographic* ordering of these new strings matches the original ordering, the *numeric* ordering does not.
- Our ***parallel rank*** algorithm works with numeric values, not string values.



000	001	01	10	11
00010100	00010111	00011011	01101001	01101110

Patricia Codes

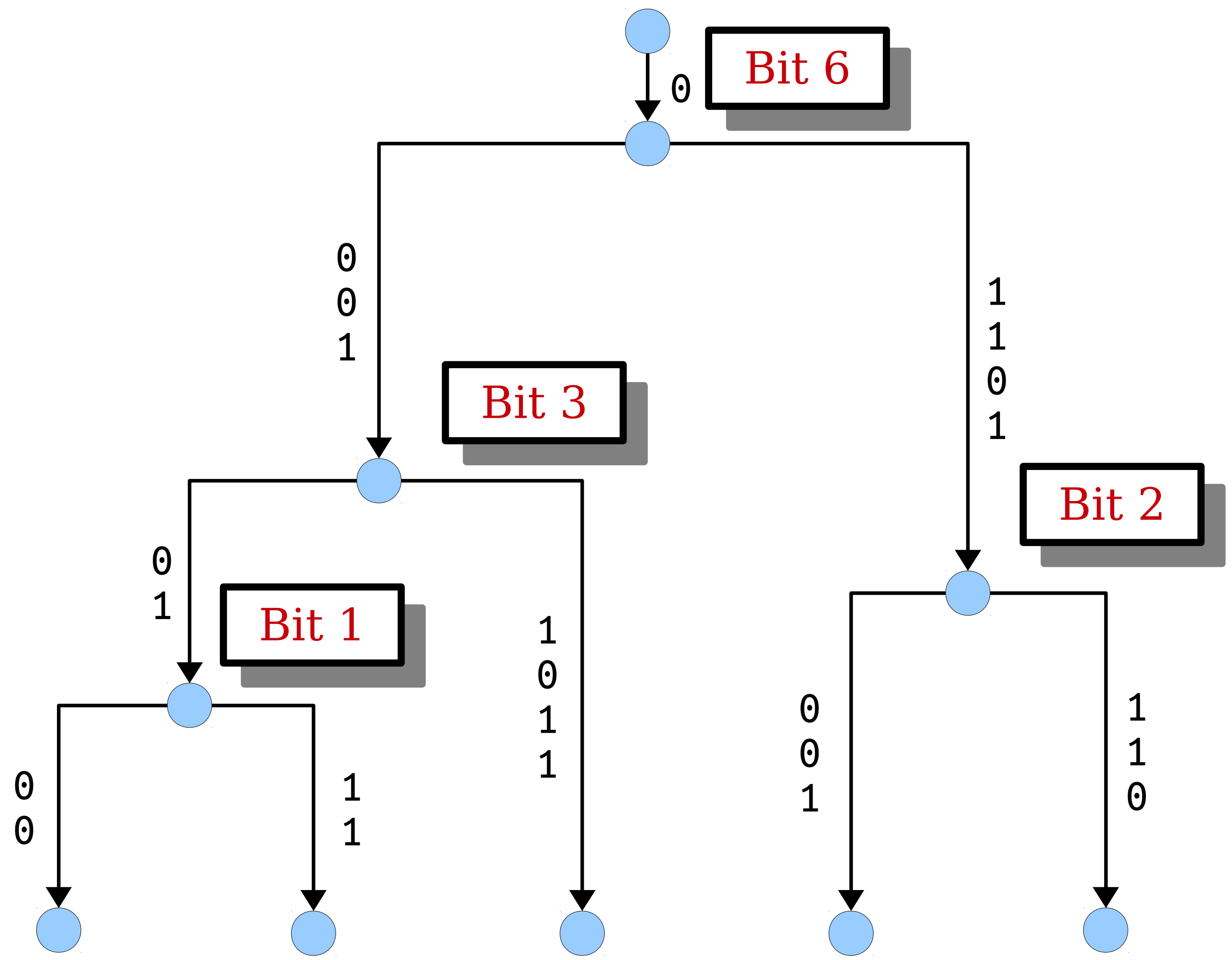
- A bit index *i* is called ***interesting*** if there is a branching node in the trie at that bit index.
- The ***Patricia code*** of an integer is the bitstring consisting of just the interesting bits in that number.



00010100	00010111	00011011	01101001	01101110

Patricia Codes

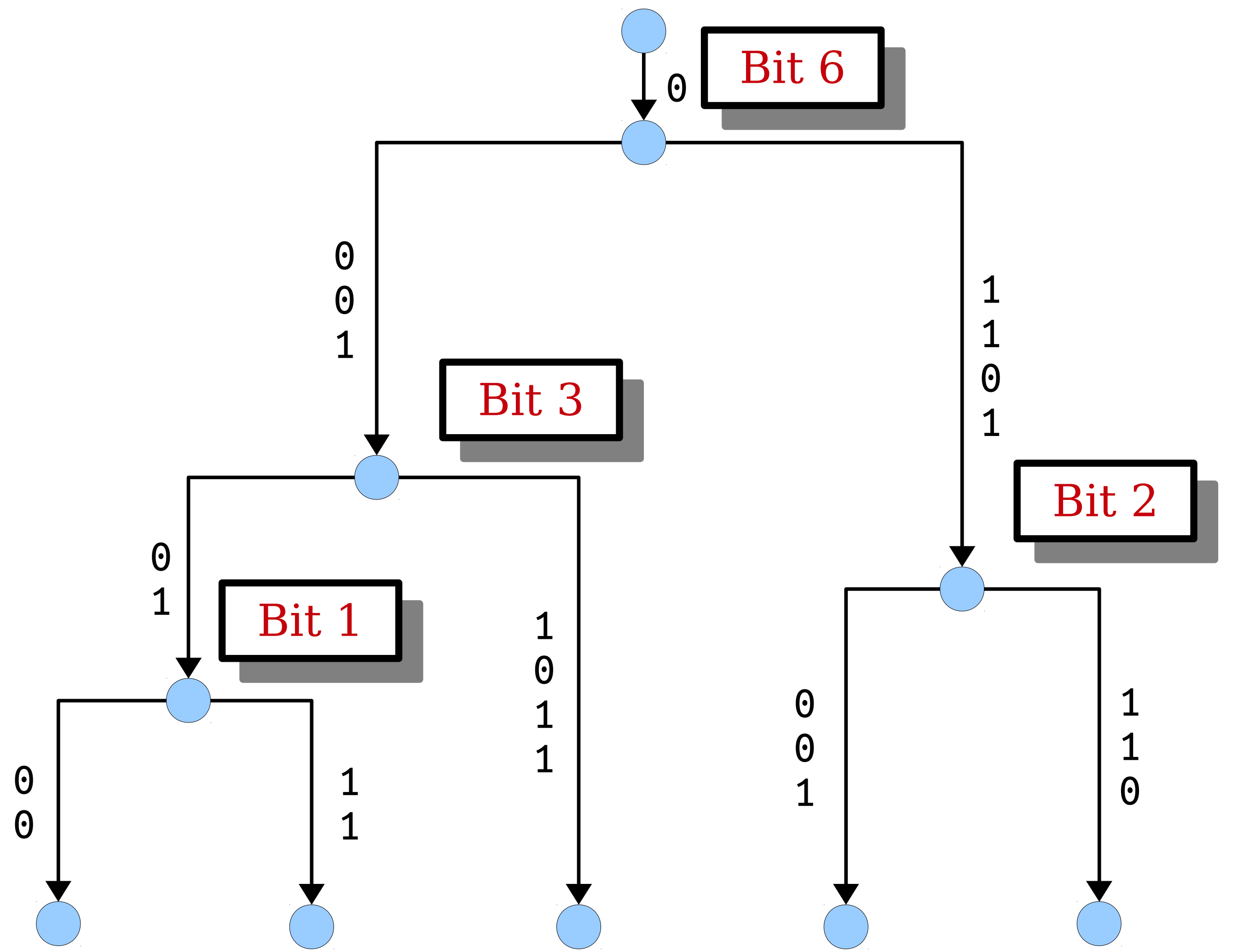
- A bit index *i* is called ***interesting*** if there is a branching node in the trie at that bit index.
- The ***Patricia code*** of an integer is the bitstring consisting of just the interesting bits in that number.



0010	0011	0101	1100	1111
0**0**01**010**0	0**0**01**011**1	0**0**01**101**1	0**1**10**100**1	0**1**10**111**0

Patricia Codes

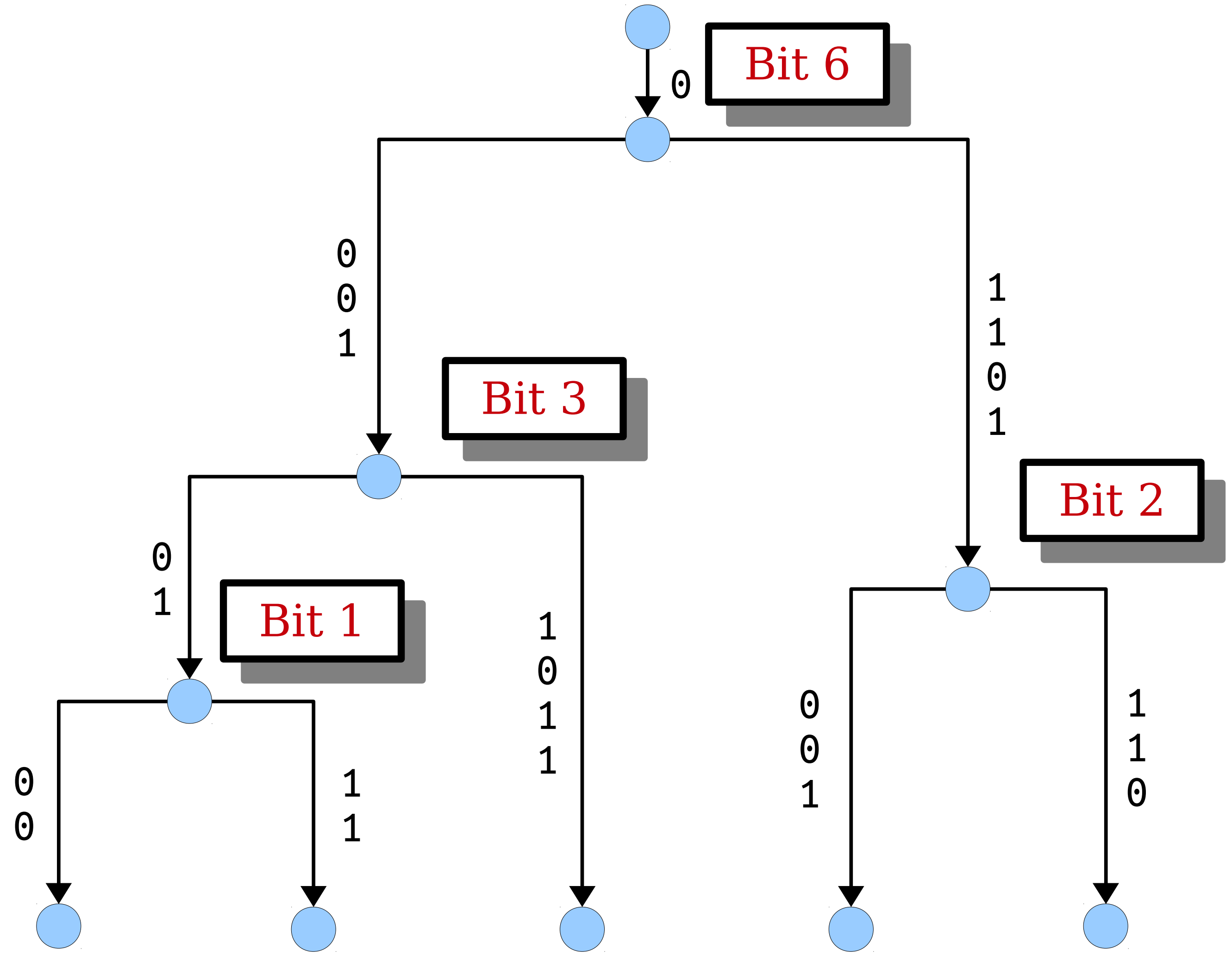
- ***Claim:*** The relative order of the integers in this trie is the same as the relative numeric order of their Patricia codes.
- Each bit either gives a direction to branch at a decision point, or is in the middle of an edge and doesn't matter.



0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Patricia Codes

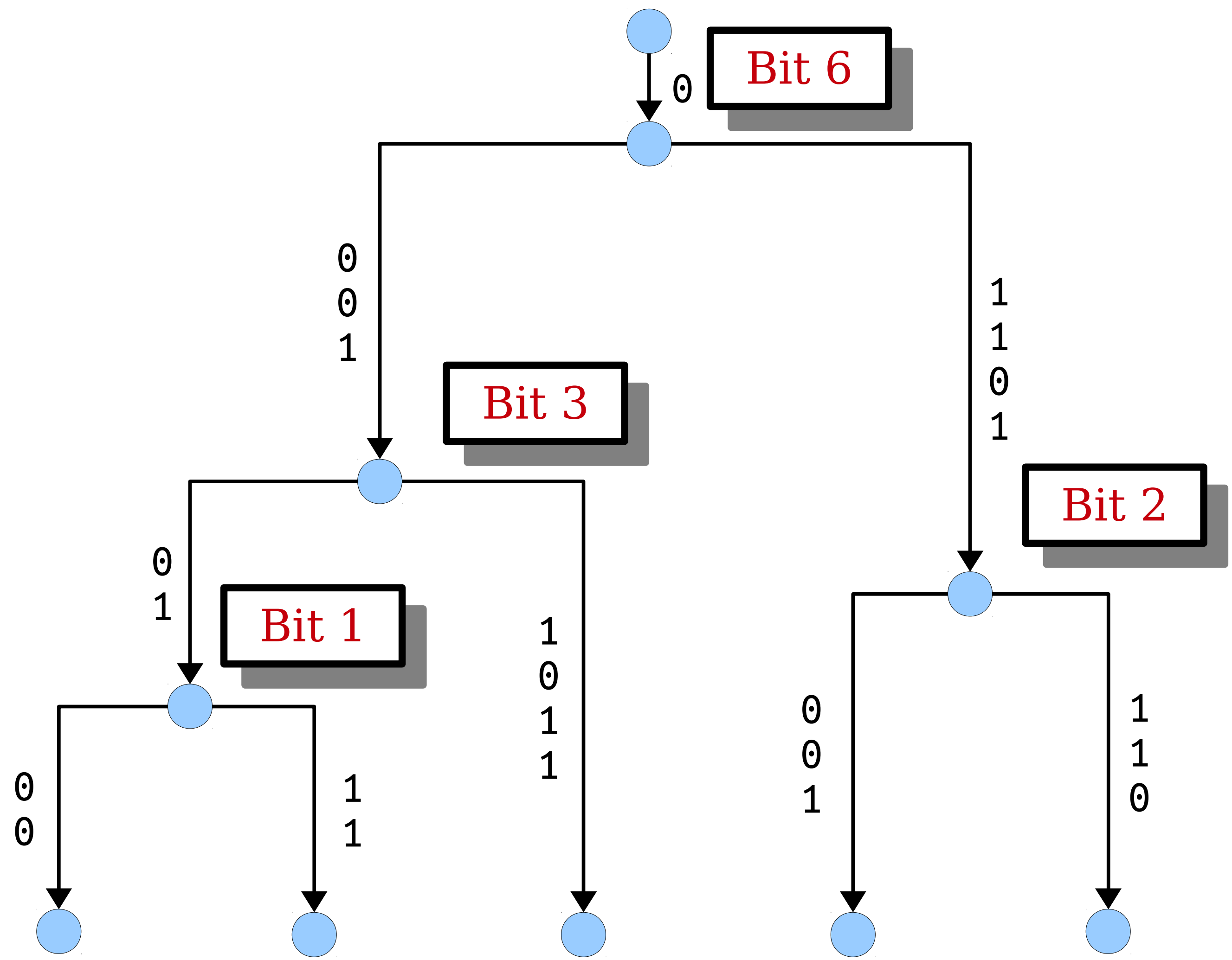
- ***Claim:*** With the right preprocessing, there's a way to (sorta) compute the Patricia code of any number in time O(1).
- We'll go over the details later today.



0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Patricia Codes

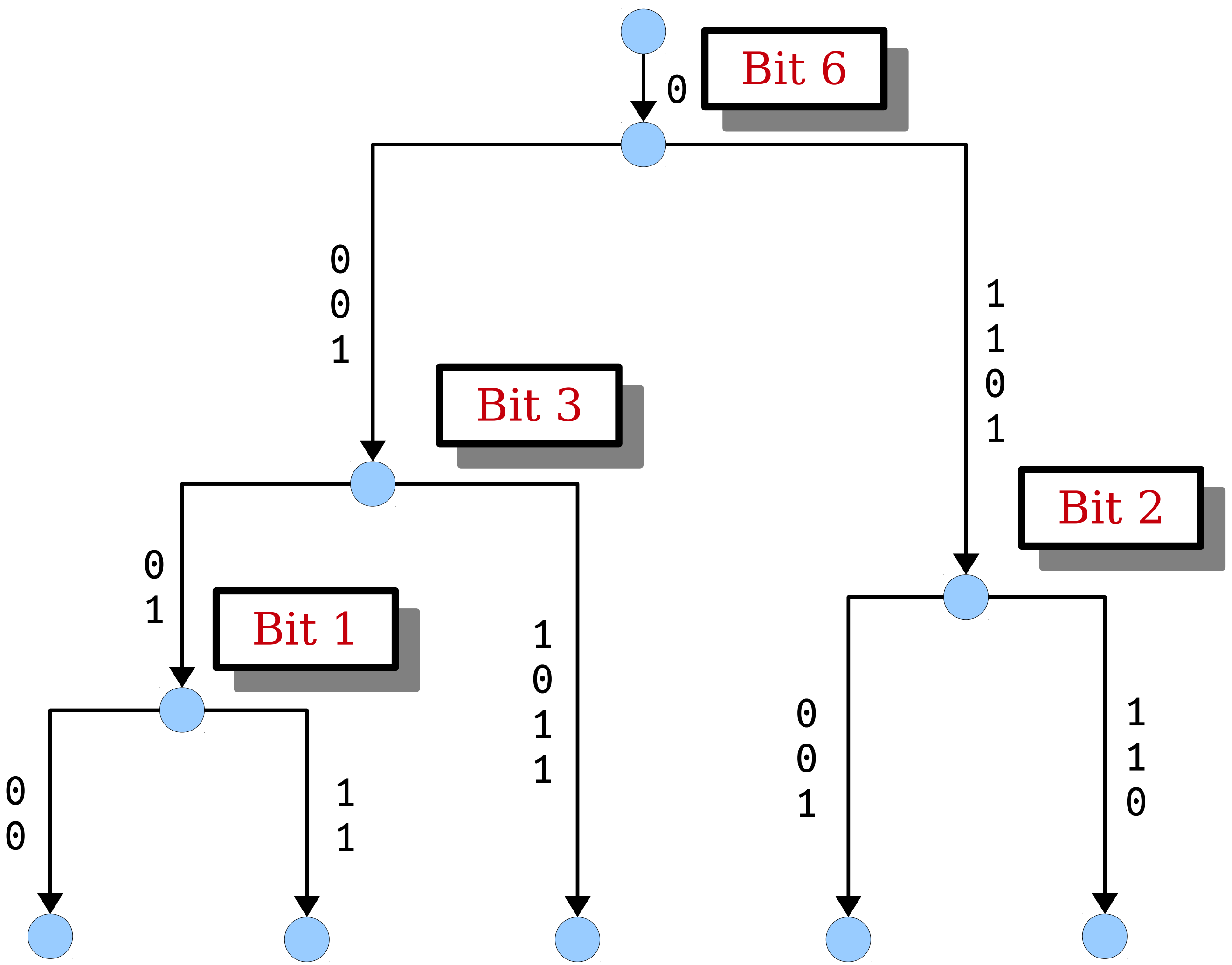
- ***Claim:*** Assuming we pick ε to be sufficiently small, the Patricia codes for our w_ε values will fit into a machine word.
- This means that we can preprocess them so that we can compute ranks of Patricia codes in time O(1).



0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Patricia Codes

- Our goal is to efficiently compute ranks among the original numbers.
- If all our Patricia codes fit into a single machine word, we can compute ***rank***(x) in time O(1), though it's a little trickier than it looks.

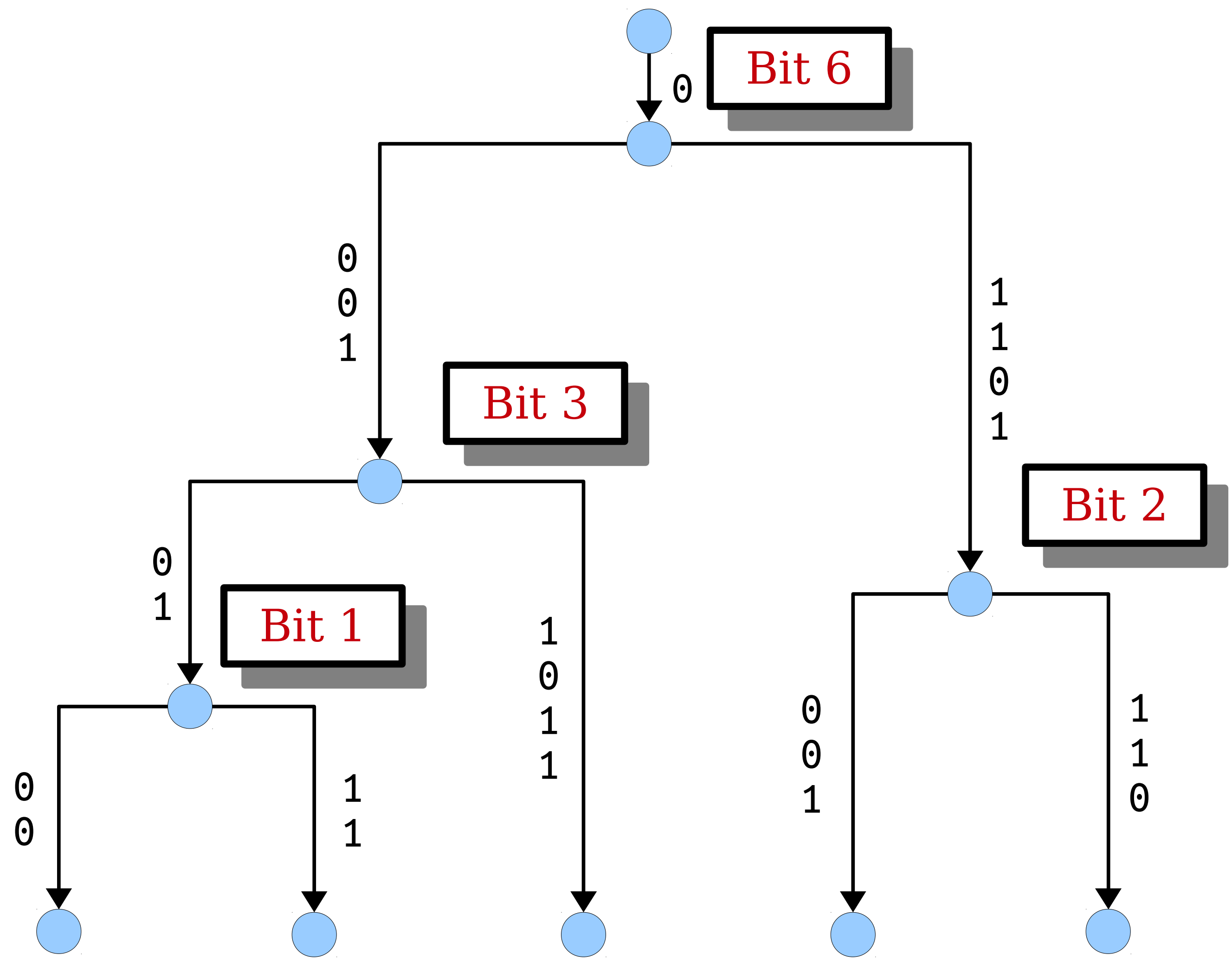


0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Computing Ranks

- Suppose we want to determine ***rank***(`00010101`).
- First, compute its Patricia code:

`00010101`

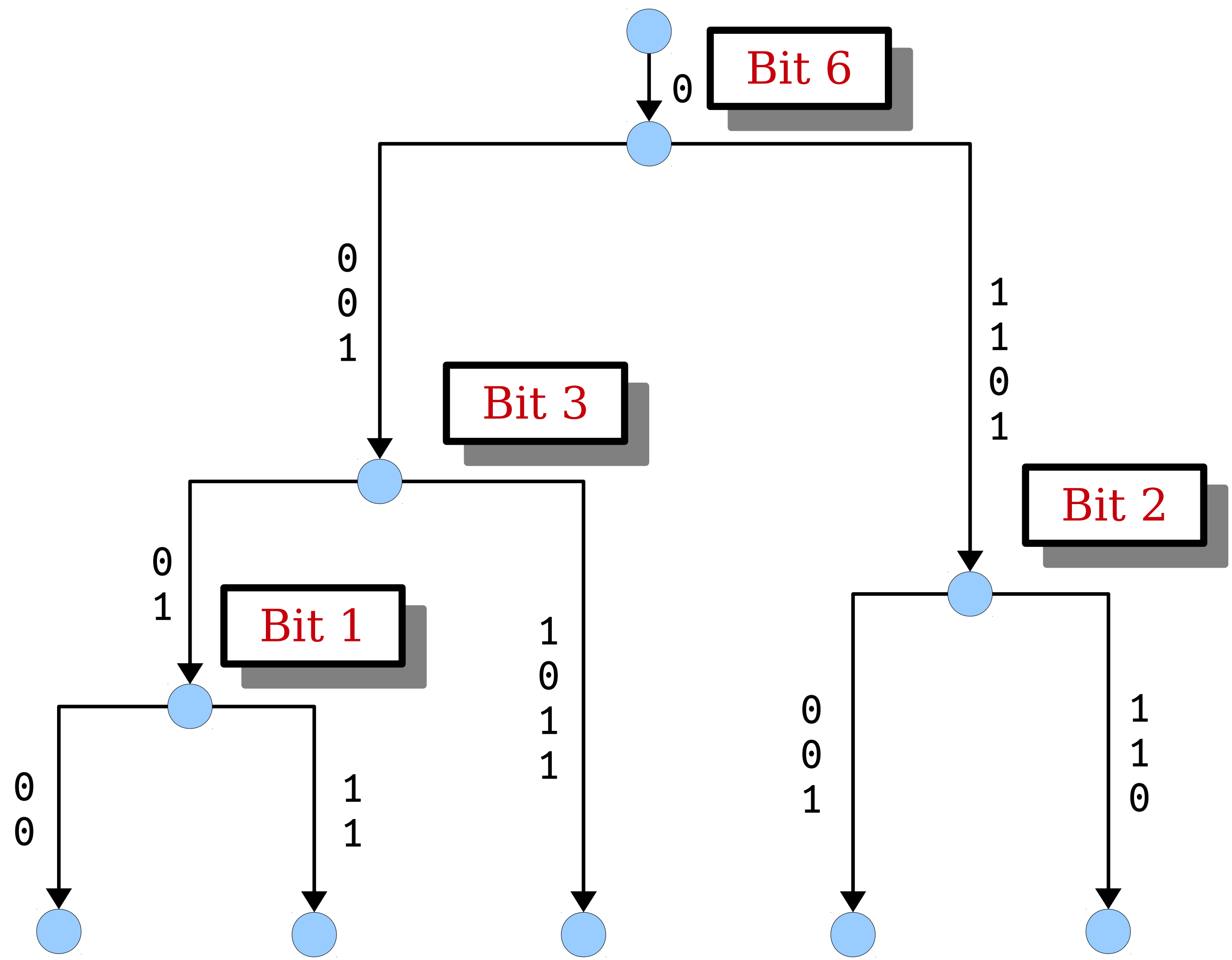


0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Computing Ranks

- Suppose we want to determine ***rank***(`00010101`).
- First, compute its Patricia code:

0010
00010101

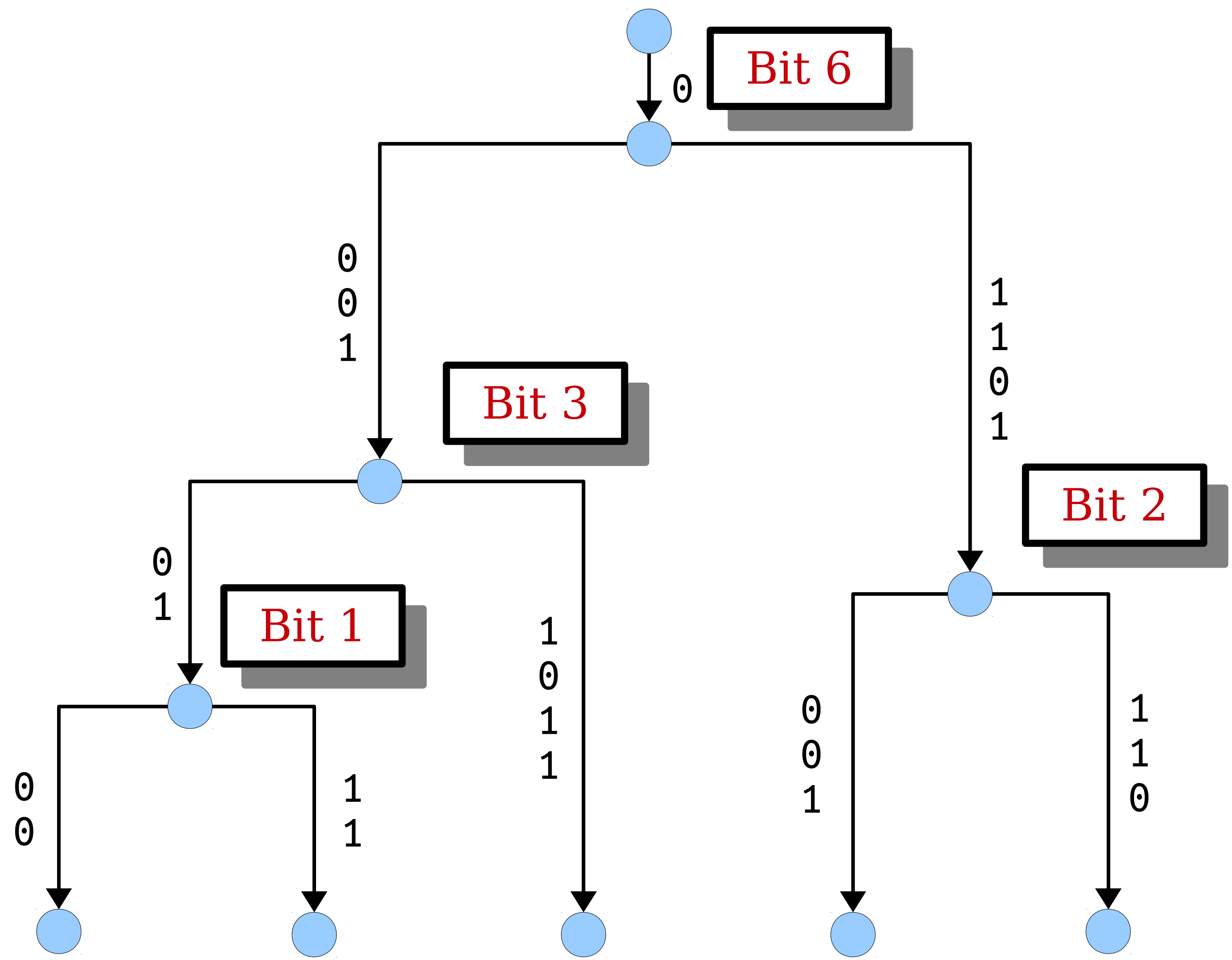


0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Computing Ranks

- Now, compute the rank of its Patricia code across the trie elements.
- Notice that the rank of this number matches the rank of its Patricia code. Cool!

0010
00010101

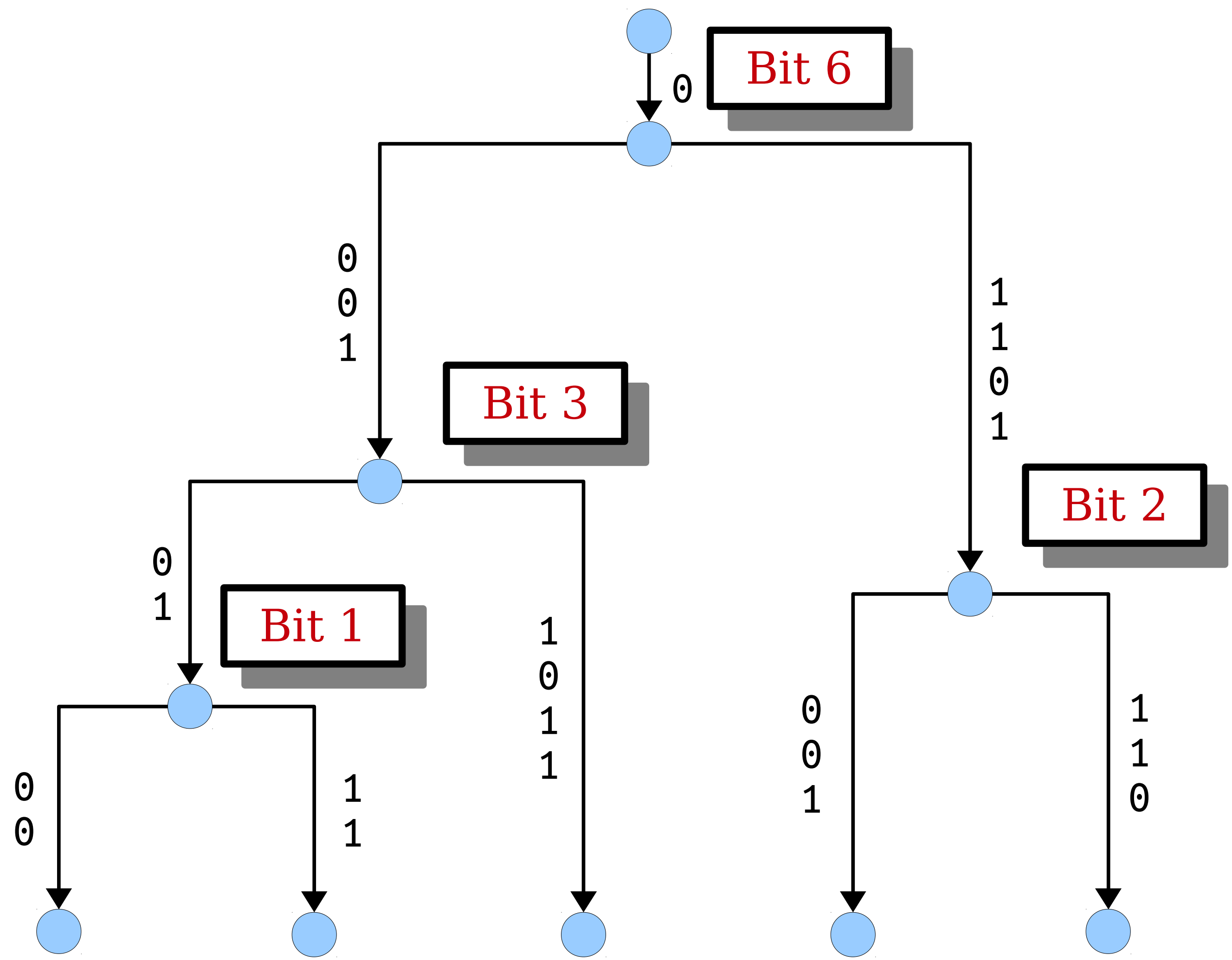


0010	**0011**	**0101**	**1100**	**1111**
00010100	00010111	00011011	01101001	01101110

Computing Ranks

- Unfortunately, things get a bit trickier here. Let's compute ***rank***(`01001110`).
- First, compute its Patricia code:

`01001110`

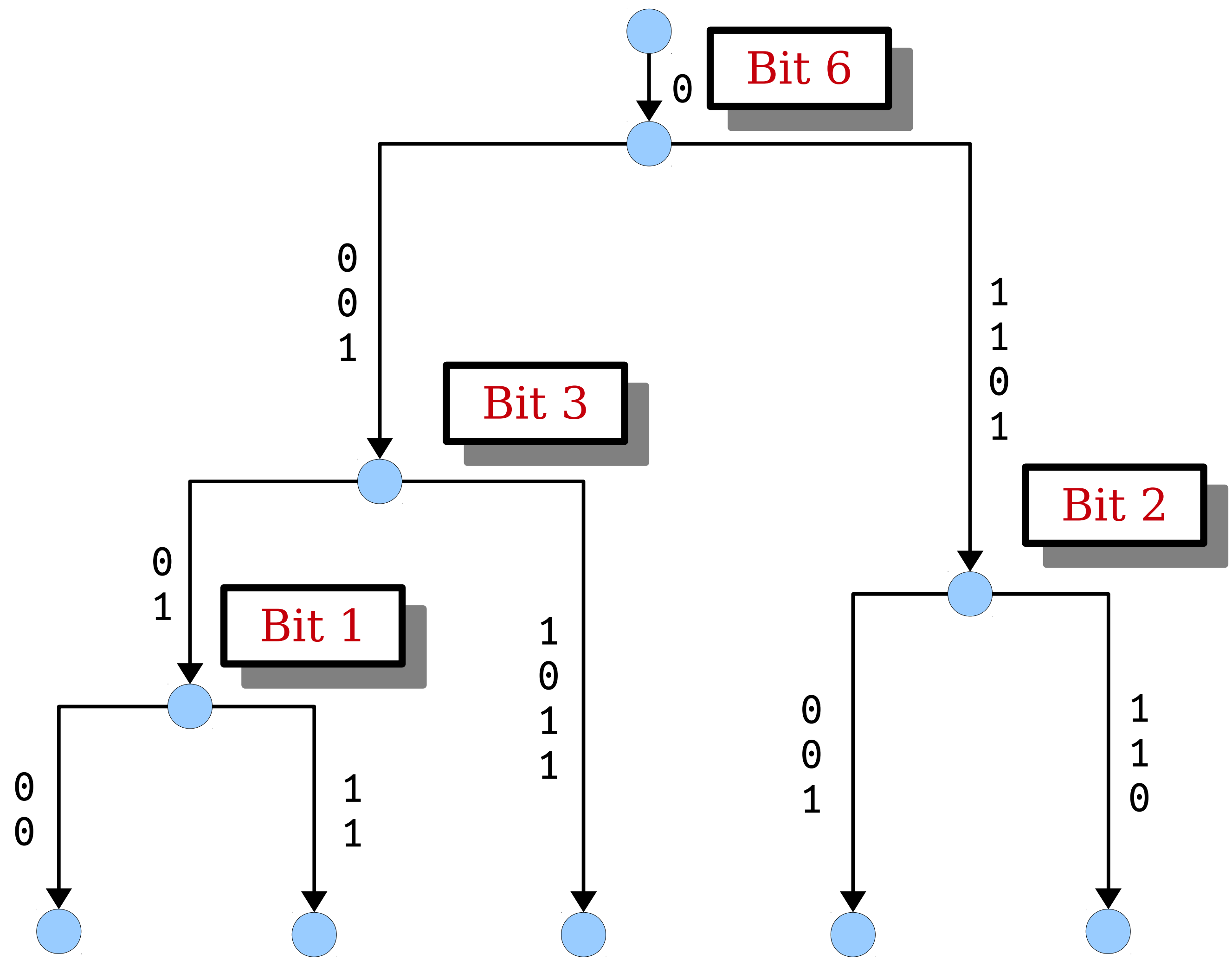


0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Computing Ranks

- Unfortunately, things get a bit trickier here. Let's compute ***rank***(`01001110`).
- First, compute its Patricia code:

1111
0<u>1</u>00<u>111</u>0

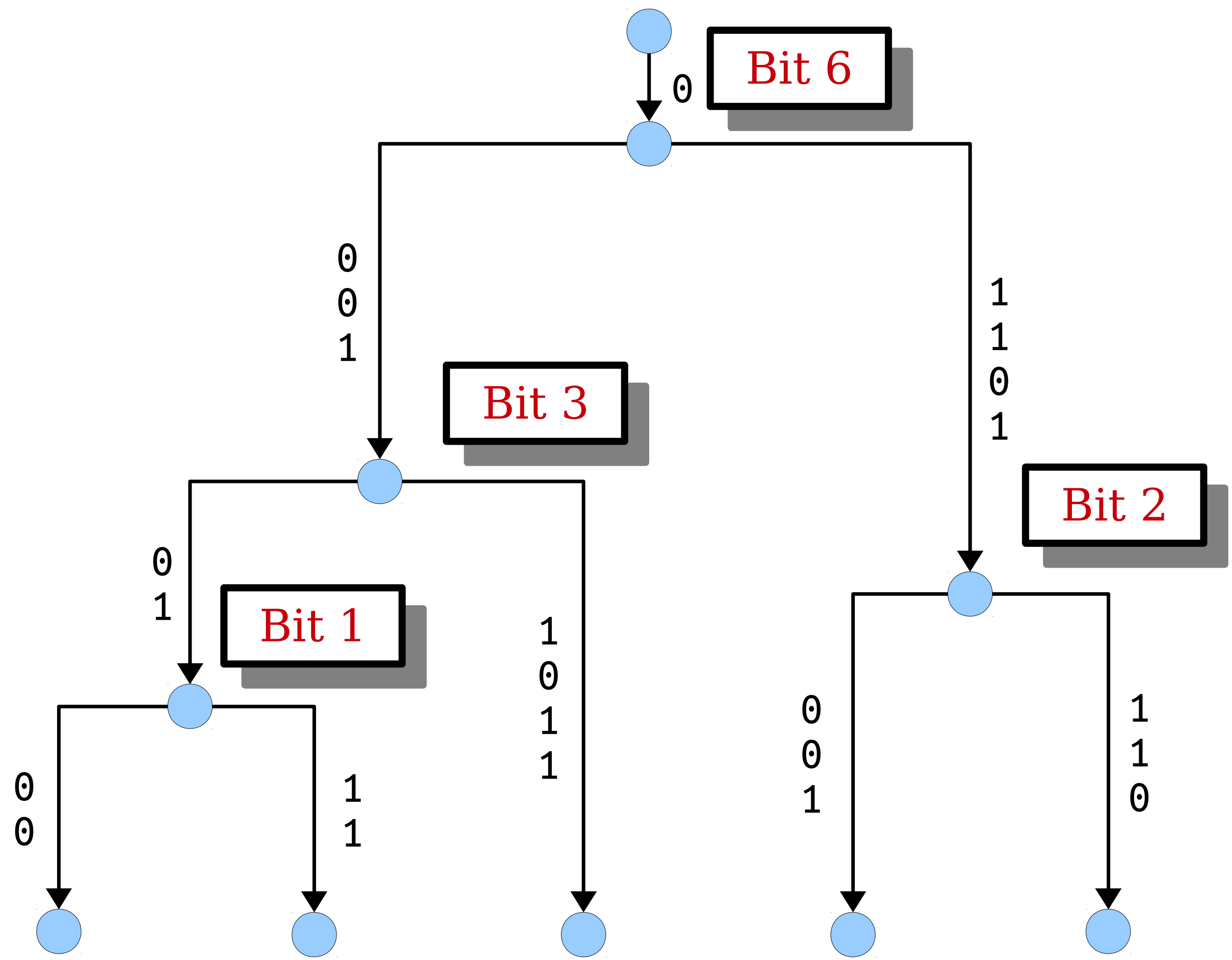


0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Computing Ranks

- Now, compute the rank of its Patricia code across the trie elements.
- Its code has rank 5, but the number itself has rank 3!
- Why did we get the wrong answer?

1111
01001110

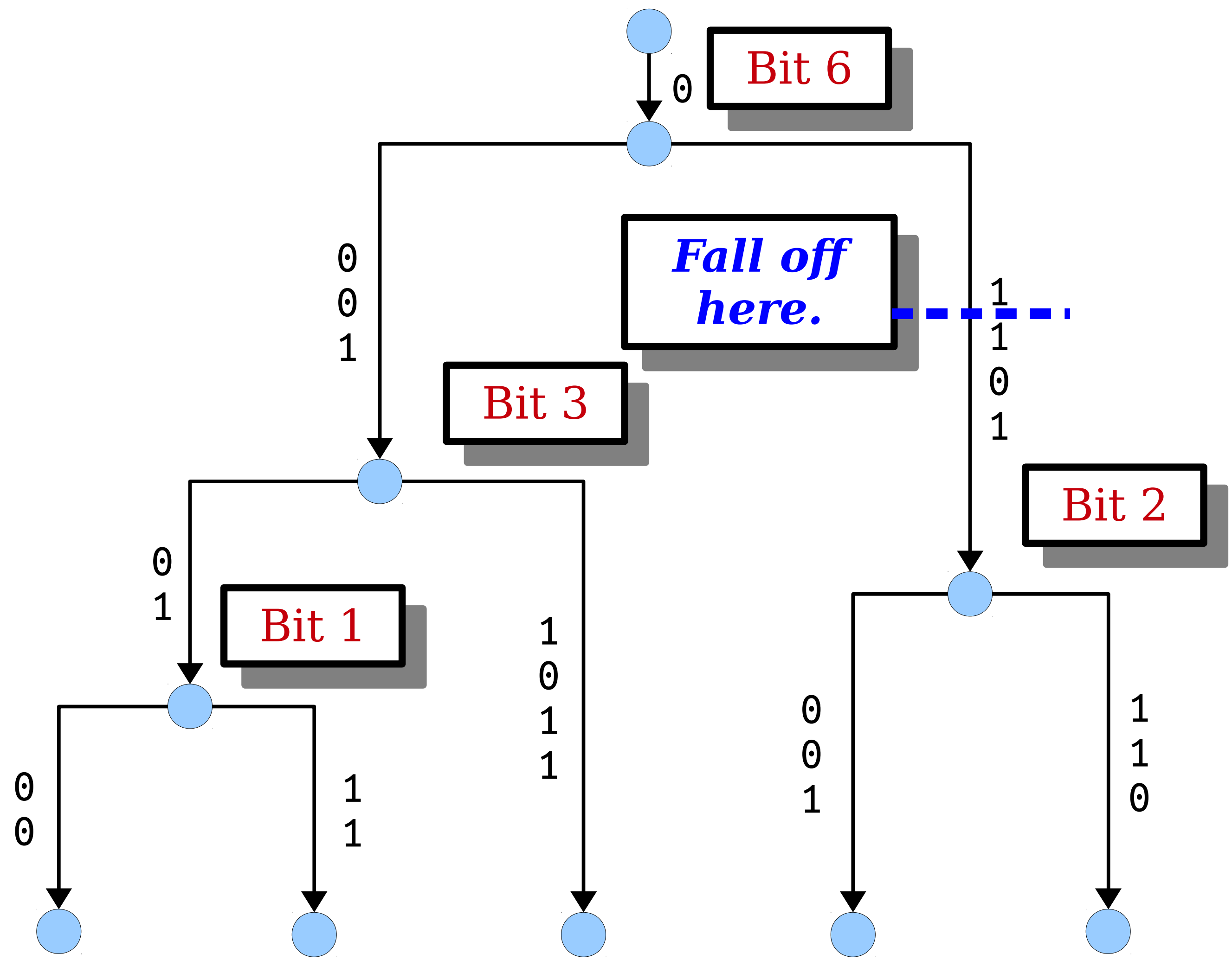


0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Computing Ranks

- Imagine we did a real, proper lookup of this key in the trie.
- Notice that we fall off the trie at the marked point.

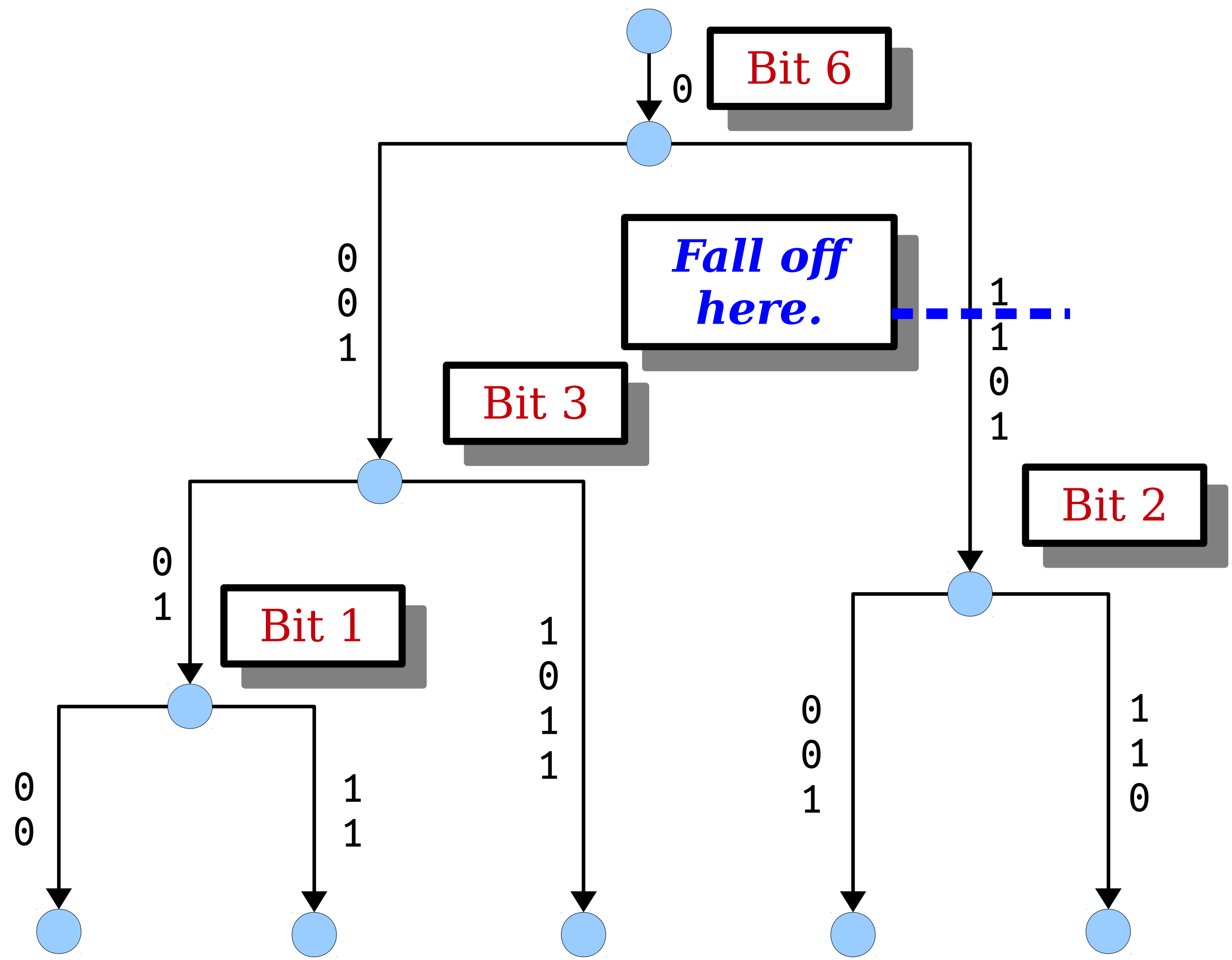
1111
01001110



0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Computing Ranks

- We made some "good" decisions followed by some "bogus" decisions.
- The good decisions are the ones where we were on the trie.
- The bogus decisions were from after we fell off.



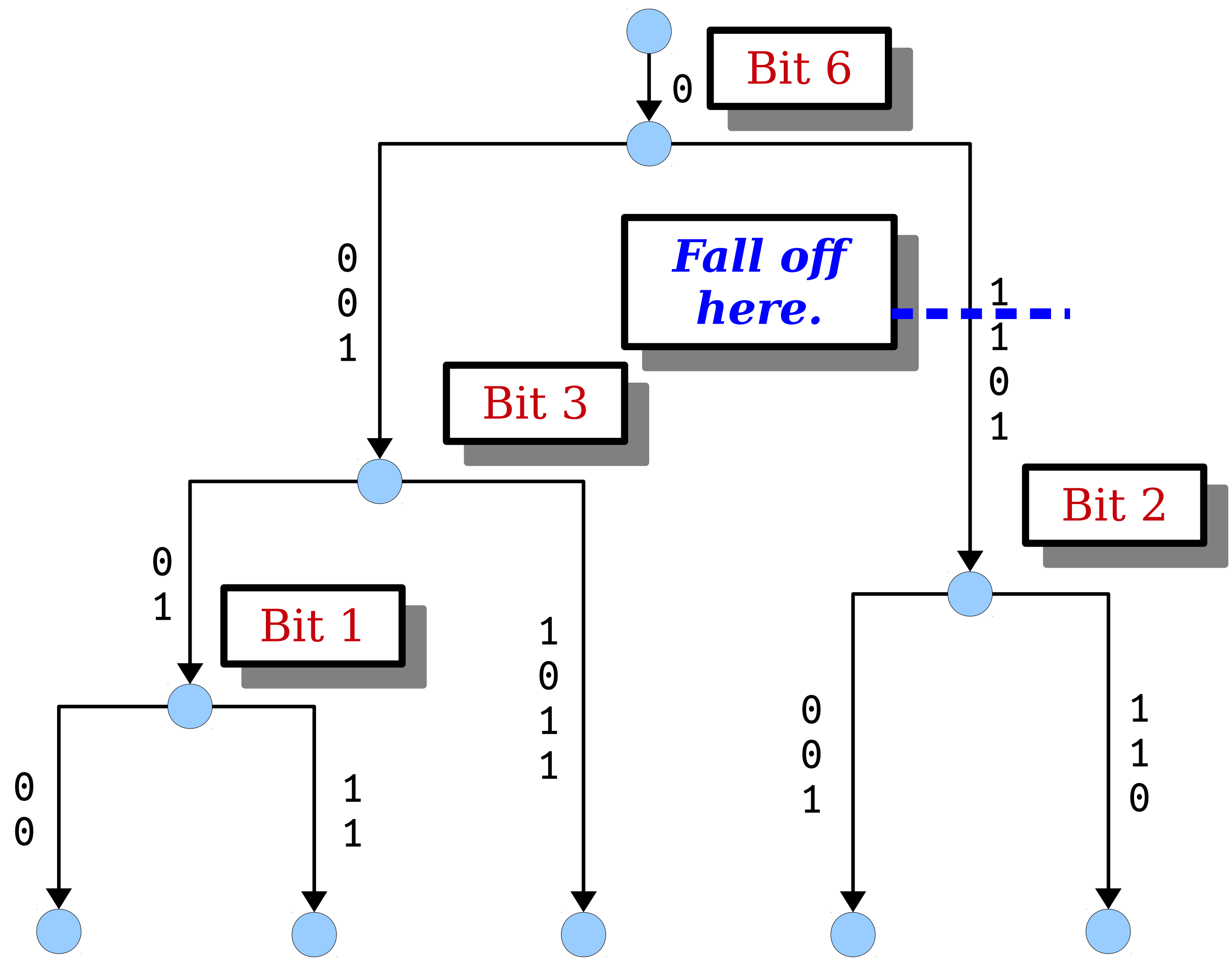
1111
01001110

0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Computing Ranks

- Look at the longest common prefix between our query key and the key next to it.
- Since the LCP has length two, we know that the first two bits of our number stayed on the trie, and then we fell off.

1111
01001110

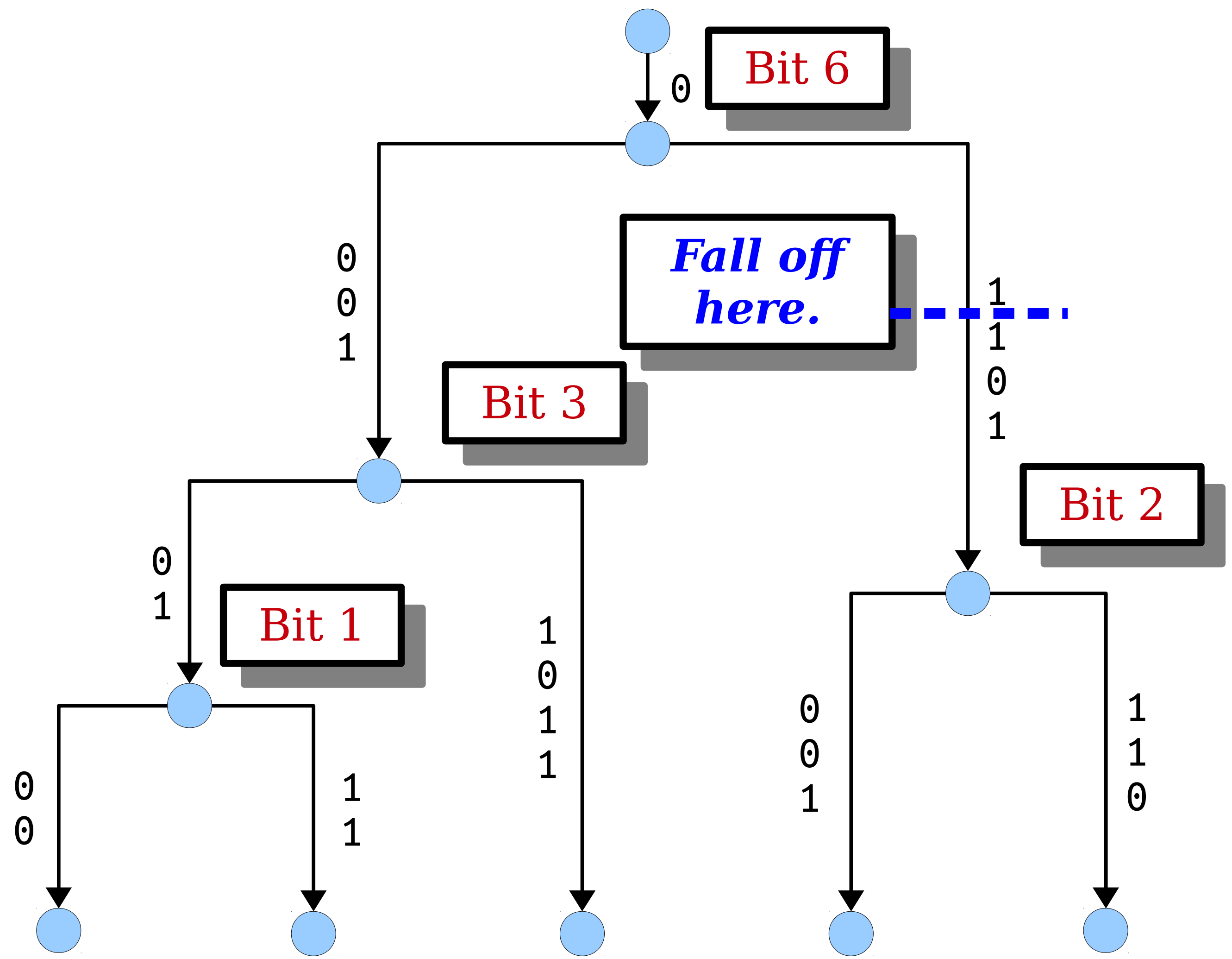


0010	0011	0101	1100	**1111**
00010100	00010111	00011011	01101001	01101110

Computing Ranks

- We fell off the trie by reading a 0.
- That means that we belong *before* everything in the subtree after that point.

1111
01001110



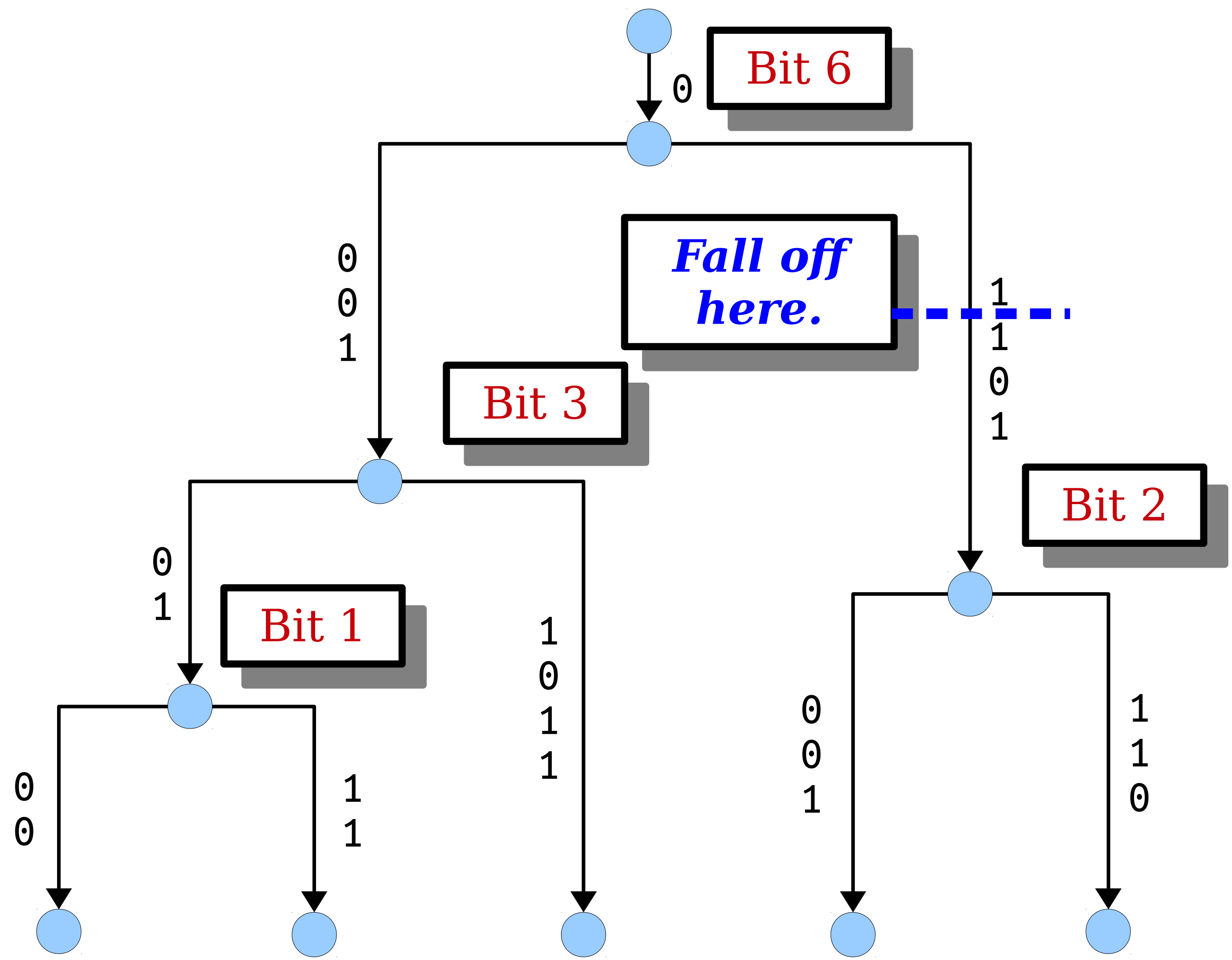
0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Computing Ranks

- ***Idea:*** Change our number to put a 0 in all positions after the mismatch, then recompute the Patricia code.
- This means "all previous comparisons are good, and then we lose on tiebreaks to everything else."

1000

01000000

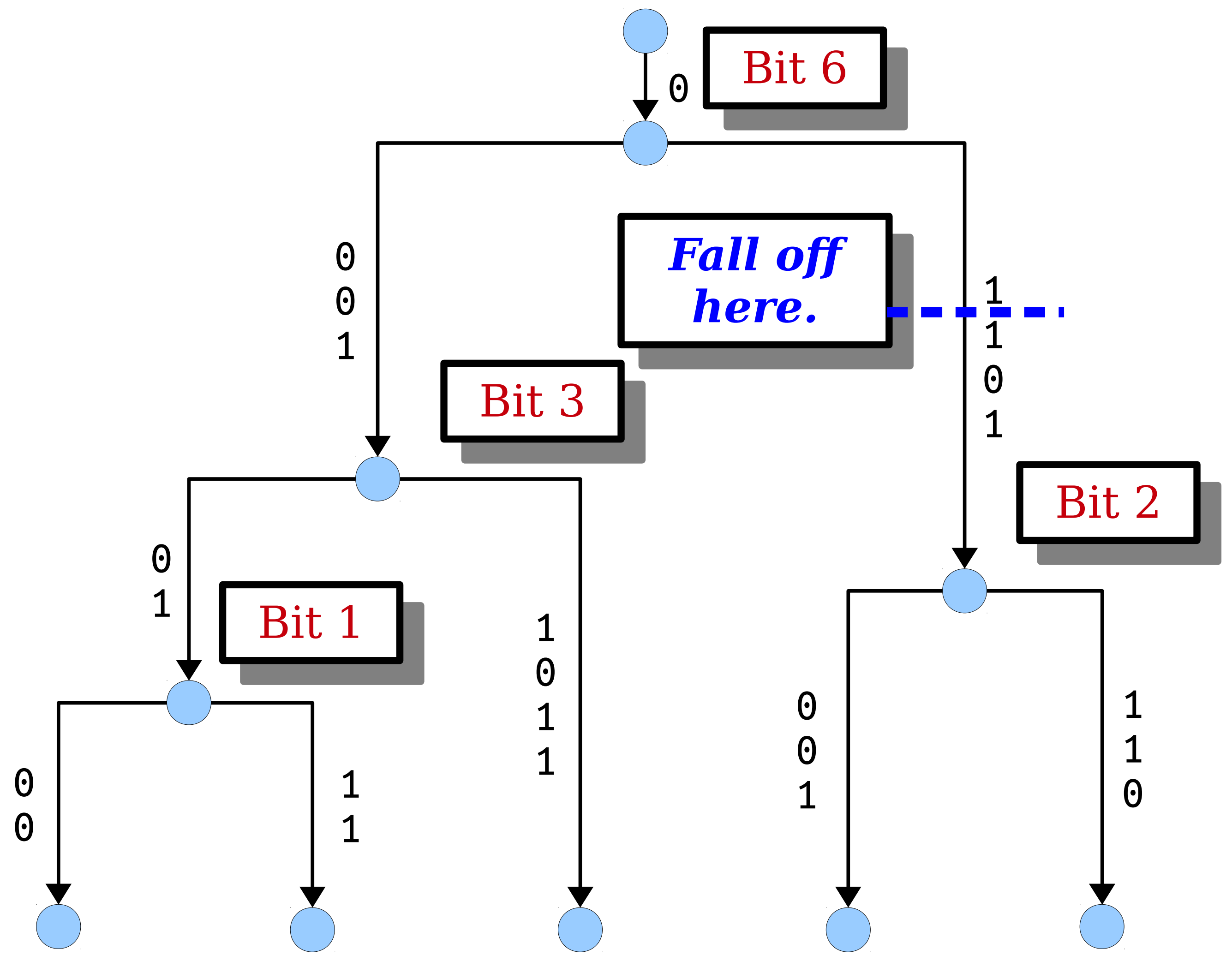


0010	0011	0101	1100	**1111**
00010100	00010111	00011011	01101001	01101110

Computing Ranks

- Let’s do a *second* rank query with this new code.
- That places us at rank 3, which is the proper position.

1000
01001110



0010	**0011**	**0101**	**1100**	**1111**
00010100	00010111	00011011	01101001	01101110

Rank in O(1)

- To search for a key:
  - Compute its Patricia code.
  - Use a ***parallel rank*** to determine the rank of its Patricia code.
  - Use our `msb` function from earlier to determine the longest matching prefix between the key and the values adjacent to it.
  - Based on the next bit, either replace all successive bits in the Patricia code either with 0s or with 1s.
  - Run a second ***parallel rank*** to determine the actual rank of the element in the sequence.
- Total cost: **O(1)**.
- I'm glossing over a few details here; check the original paper for details.

Time-Out for Announcements!

Midterm Exam

- The midterm is tonight!
  - It’s from **7PM – 10PM**.
  - It’s in **Hewlett 200**.
- You get a single, double-sided sheet of 8.5” × 11” notes with you during the exam.
- ***Go rock this exam***. You’re all awesome. Show us how much you’ve learned.

Final Project Presentations

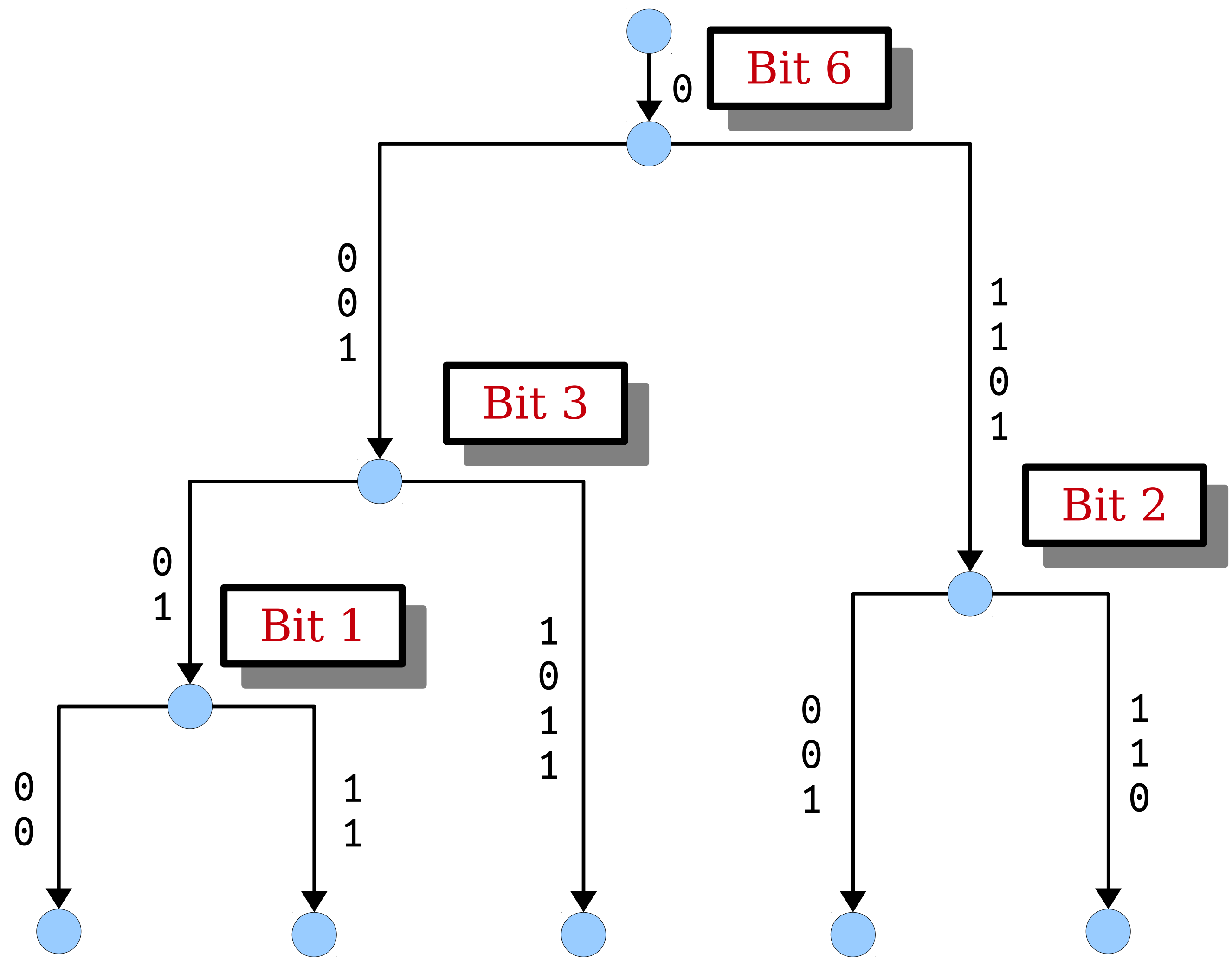
- We’ve just about finished getting time slot signups from each team.
- Once that schedule is ready, we’ll post it to the course website.
- Speaking from experience – these presentations will be a lot of fun. Feel free to pick a few that look interesting and to stop on by!

Back to CS166!

Implementing this Idea

Implementing this Idea

- We now have a clever approach for compressing keys based on Patricia tries.
- In this discussion, I've drawn the actual trie off to the side here.
- We used this trie to determine where the "interesting" bits were.

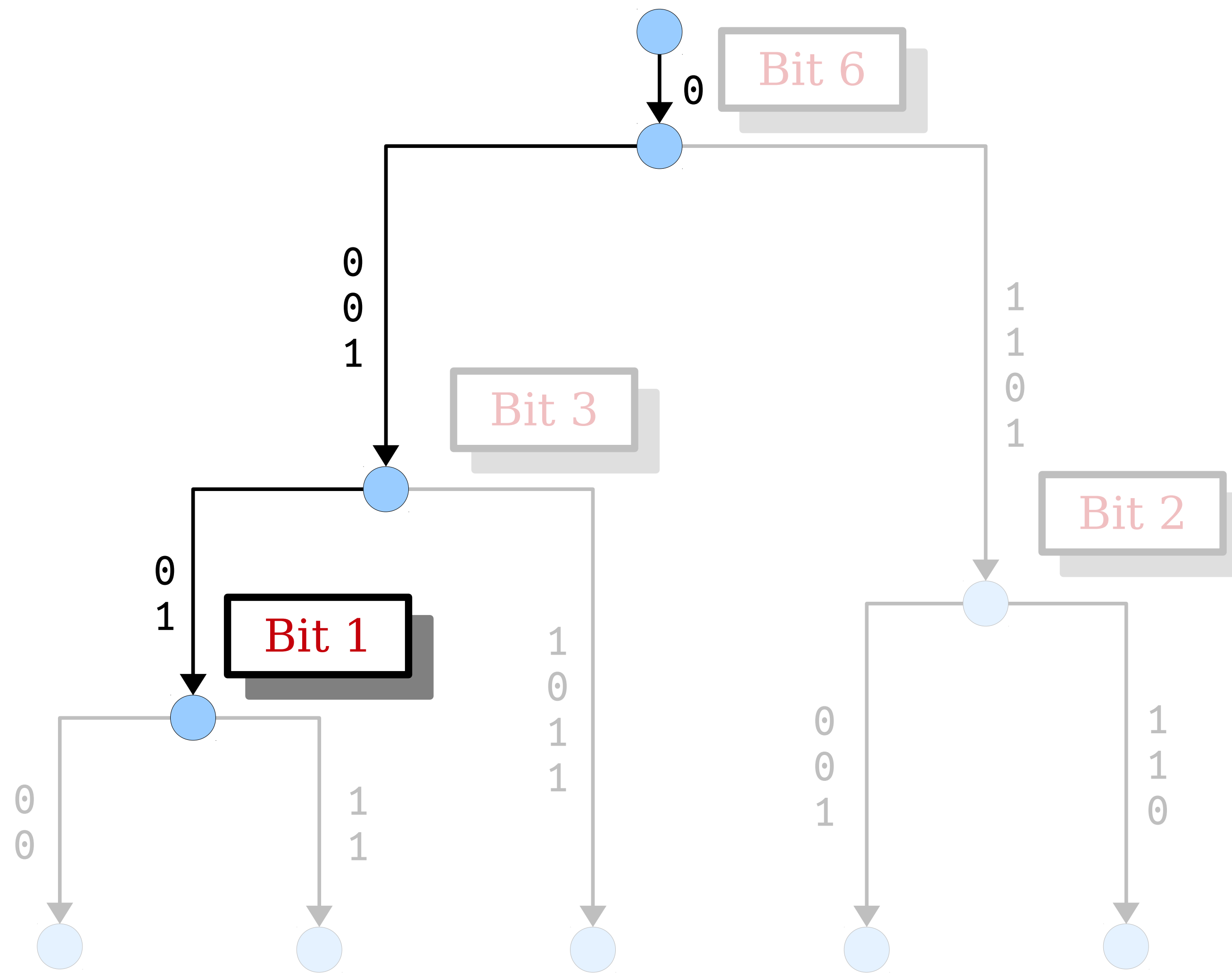


0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Implementing this Idea

- We can find all the interesting bits in a collection of keys without actually building this trie.
- ***Idea:*** There's a connection between branching nodes in the trie and the `lcp`'s of the keys.

00010100
00010111

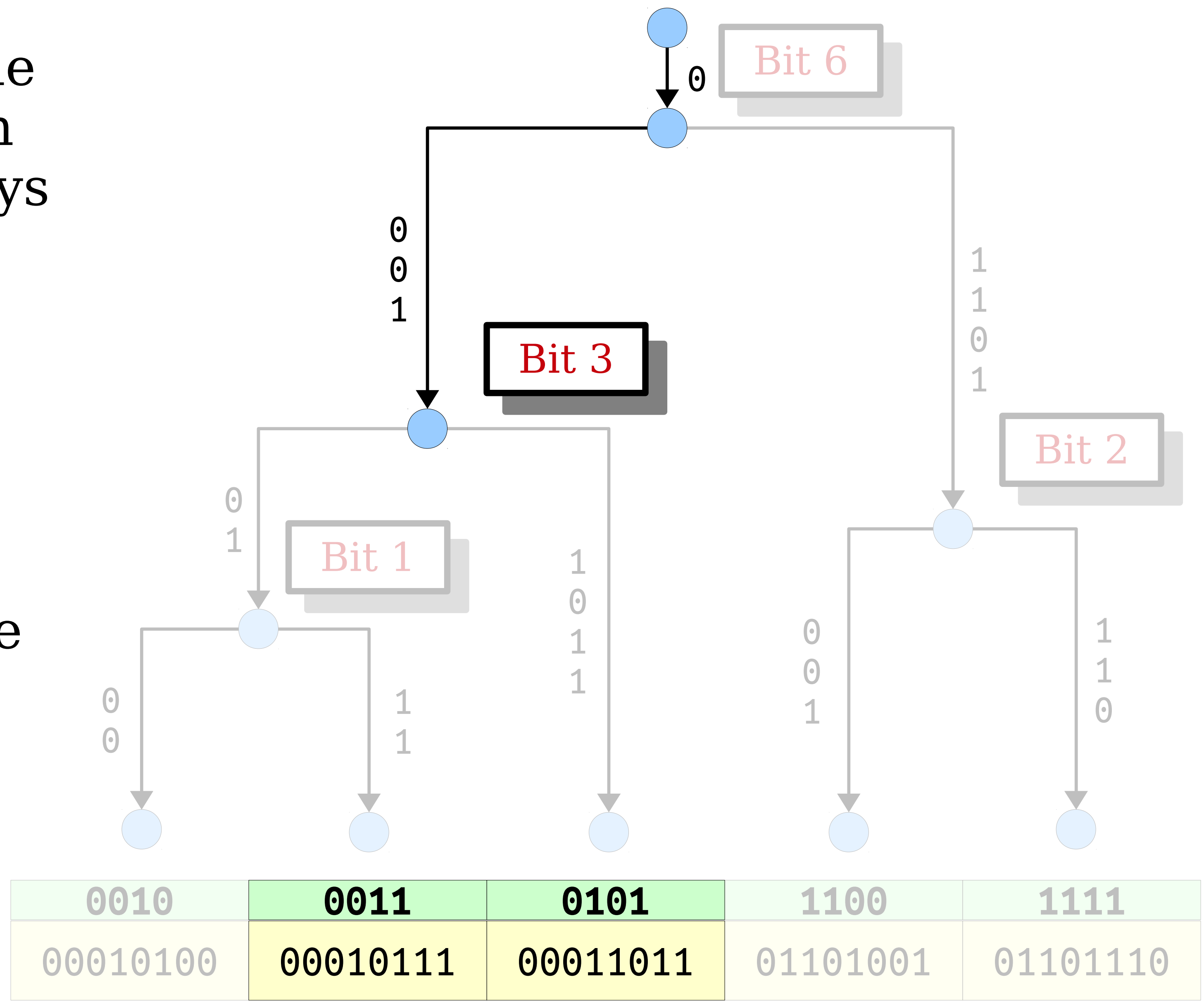


0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Implementing this Idea

- We can find all the interesting bits in a collection of keys without actually building this trie.
- ***Idea:*** There's a connection between branching nodes in the trie and the `lcp`'s of the keys.

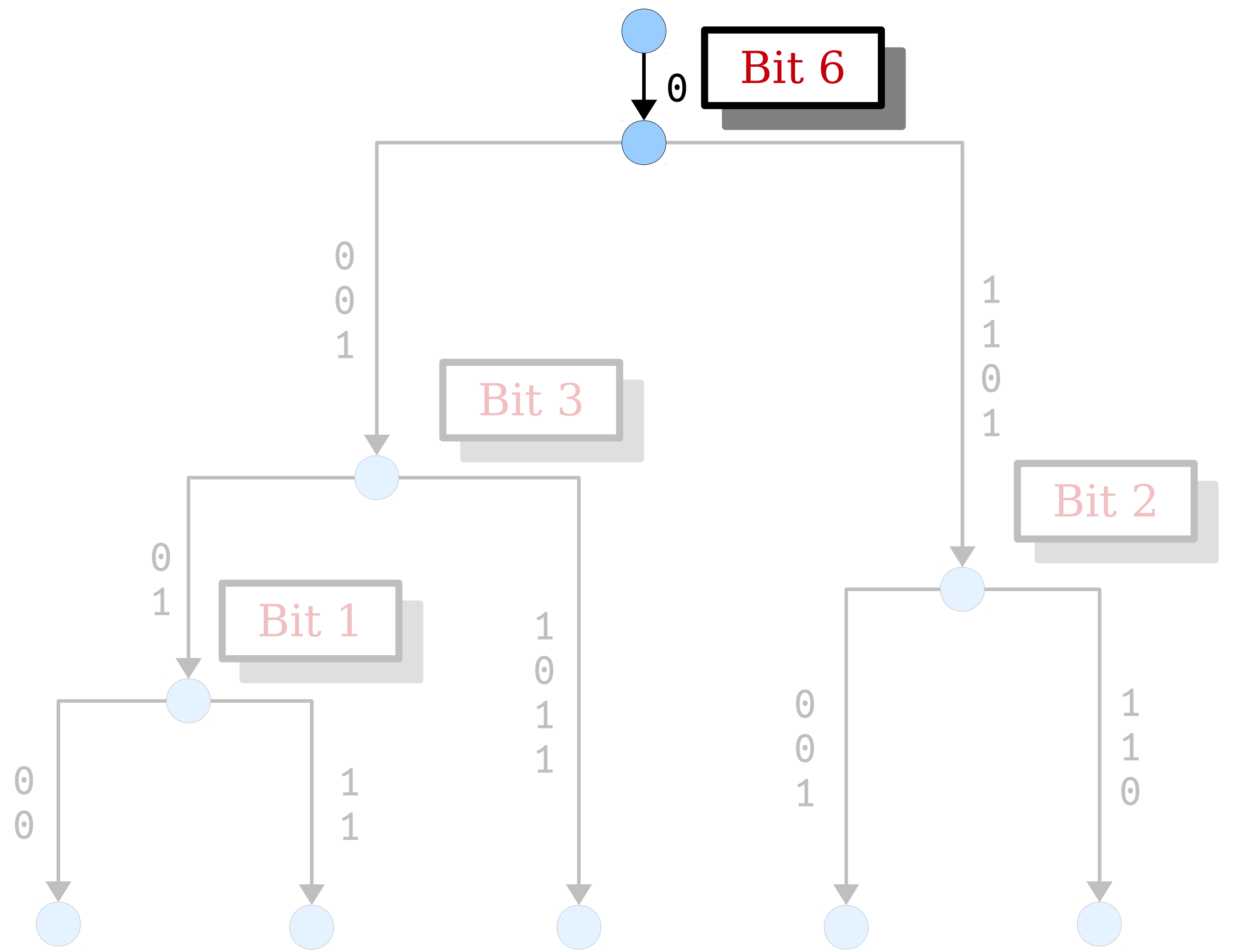
00010111
00011011



Implementing this Idea

- We can find all the interesting bits in a collection of keys without actually building this trie.
- ***Idea:*** There's a connection between branching nodes in the trie and the `lcp`'s of the keys.

00011011
01101001

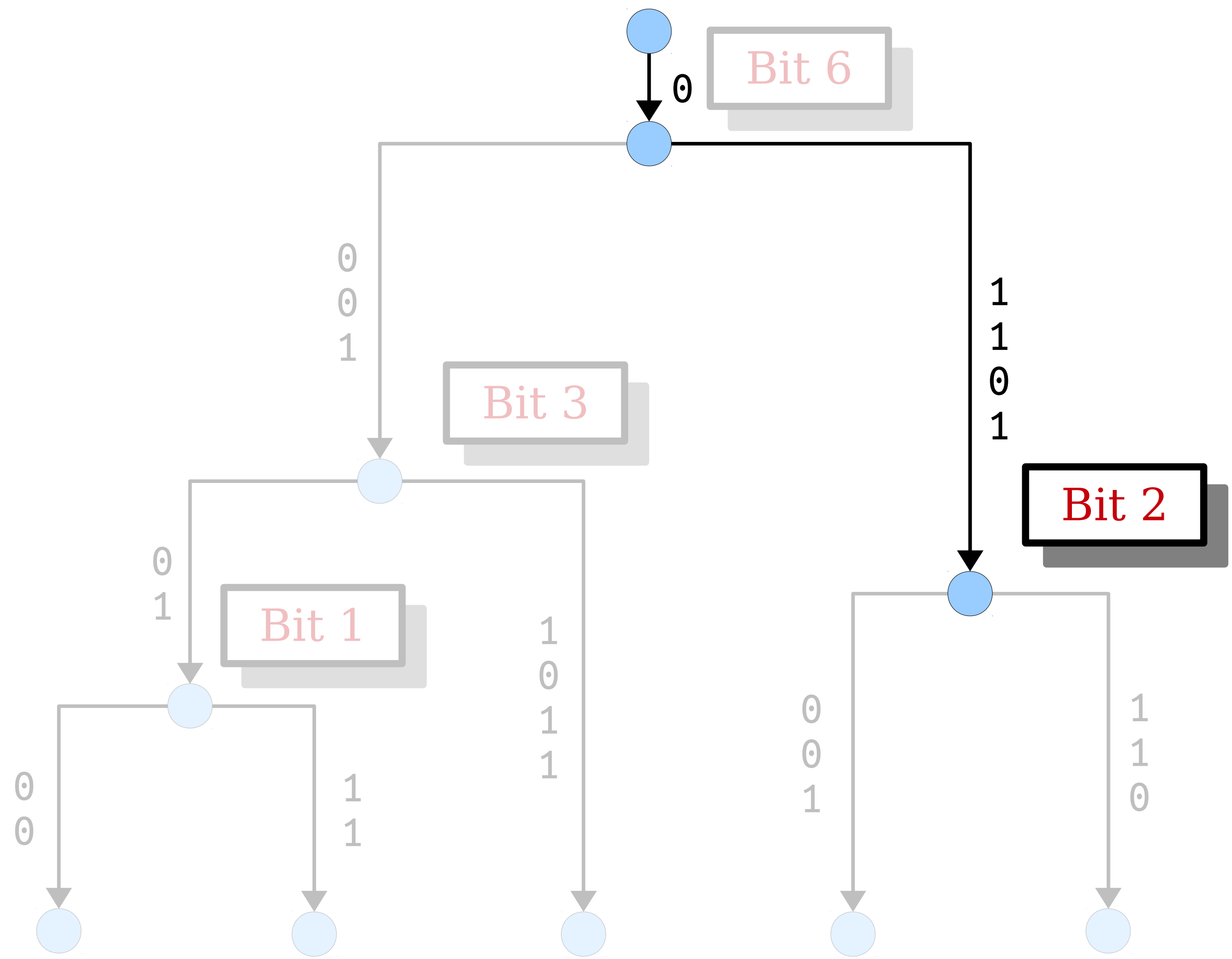


0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Implementing this Idea

- We can find all the interesting bits in a collection of keys without actually building this trie.
- ***Idea:*** There's a connection between branching nodes in the trie and the `lcp`'s of the keys.

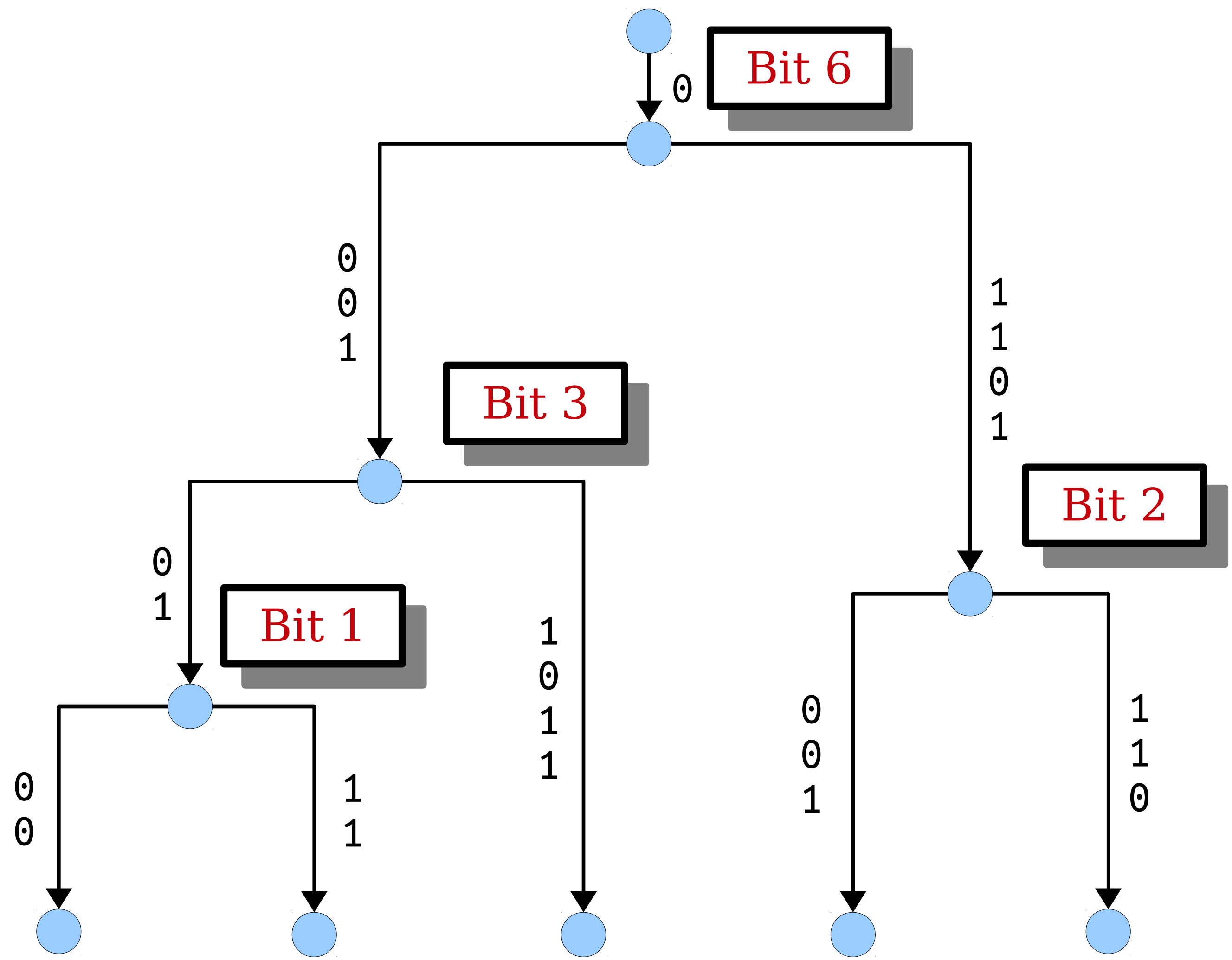
01101001
01101110



0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Implementing this Idea

- Since we don't need the Patricia trie, we can cast it off into the luminiferous aether.
- We can just store the indices of the interesting bits and the Patricia codes of the keys.



0010	**0011**	**0101**	**1100**	**1111**
00010100	00010111	00011011	01101001	01101110

Implementing this Idea

- Since we don't need the Patricia trie, we can cast it off into the luminiferous aether.
- We can just store the indices of the interesting bits and the Patricia codes of the keys.

0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Implementing this Idea

- We've assumed up to this point that we can compute Patricia codes in time O(1).
- This is the last step we need to figure out!
- How do we do this?

0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Extracting Patricia Codes

- We'd like to extract the w^{ε} interesting bits from each machine word, and ideally, to do so quickly.
- We can start by building up a bitmask to mask everything except those interesting bits.
- If we can compact these bits together, we've got the Patricia code!

```
  00011010 01101110 01111000 01001101 00101111 00001101 01110111 01100001
∧ 00010000 00000000 00010100 00000000 00010000 00000010 00000000 00000001
  00010000 00000000 00010000 00000000 00000000 00000000 00000000 00000001
```

Extracting Patricia Codes

- We now have all the bits we want, but they're spread apart too far.
- We saw last time that by using multiplication by an appropriate constant, we can compact bits together.

```
a0000000b0000000c0000000d0000000
```

```
  a0000000b0000000c0000000d0000000000000000000000000000
         a0000000b0000000c0000000d0000000000000000000000
                a0000000b0000000c0000000d000000000000000
+                      a0000000b0000000c0000000d0000000
-----------------------------------------------------
  ?????????????????????abcd????????????????????????????
```

Extracting Patricia Codes

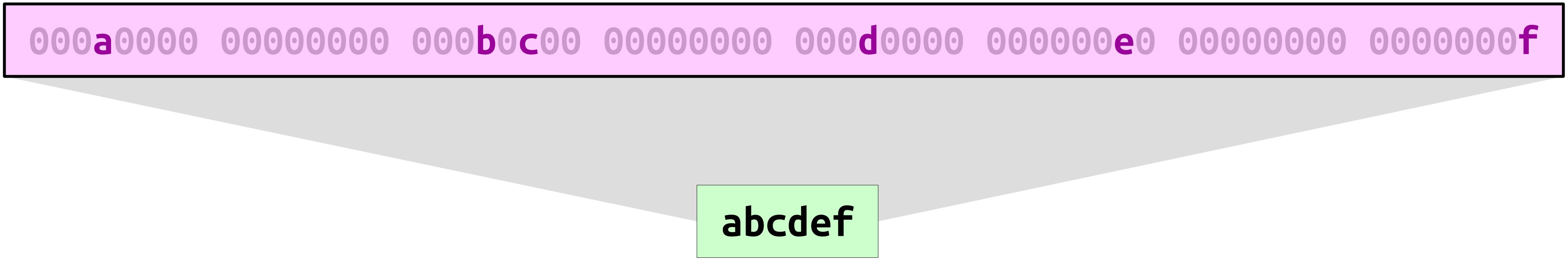
- The approach we used last time worked well because we knew those bits were evenly-spaced.
- ***Problem:*** Our “interesting” bits aren’t well-spaced across the word in question.
- This may make it impossible to get all the bits next to one another purely using a clever multiplication.

`a000b000c000d000e000f000g000h000ijkl`

- Fortunately, there’s an escape hatch.

Approximate Patricia Codes

- Patricia codes are useful because they
  - contain enough information to compute ranks, and
  - compact that information into a small space.
- ***Idea:*** Maintain the second property by doing a “decent” job compacting bits, rather than a “perfect” job.



Approximate Patricia Codes

- Patricia codes are useful because they
  - contain enough information to compute ranks, and
  - compact that information into a small space.
- ***Idea:*** Maintain the second property by doing a “decent” job compacting bits, rather than a “perfect” job.

`000a0000 00000000 000b0c00 00000000 000d0000 000000e0 00000000 0000000f`

`a00000b00c0d00e000000f`

Approximate Patricia Codes

- An ***approximate Patricia code*** is a bitstring containing all the interesting bits of a number in the same relative order, with some extra 0's deterministically interspersed.
- ***Claim:*** We can use approximate Patricia codes rather than true Patricia codes to compute ranks. The relative orders of the codes will come back the same.

`000a0000 00000000 000b0c00 00000000 000d0000 000000e0 00000000 0000000f`

`a00000b00c0d00e000000f`

Approximate Patricia Codes

- ***Theorem:*** Suppose we have a w^{ε} interesting bits. Then there is a way to compute a multiplier M, a mask K, and a shift S such that

$$((n \times M) \gg S) \mathbin{\&} K$$

  is an approximate Patricia code for n that uses $w^{4\varepsilon}$ bits, and these values can be computed in time $O(w^{4\varepsilon})$.
- ***Proof:*** Some very clever arguments involving induction and modular arithmetic. Check Fredman and Willard's paper for details!
- ***Challenge:*** Find a simple, visual, intuitive explanation for this algorithm.

Closing In on Fusion Trees

- Our goal is to build a data structure that holds w^{ε} integers with w bits each in a way that supports ***rank*** in time O(1).
- Given w^{ε} integers, we can do some preprocessing to form $w^{4\varepsilon}$-bit approximate Patricia codes for them.
- Storing those approximate codes requires $w^{5\varepsilon}$ bits.
- ***Observation:*** Suppose we pick $\varepsilon = {}^{1}/_{6}$. Then we can store all of those codes in a single machine word!

What is $w^{1/6}$ on a real computer?

We have a ways to go before this strategy will have any chance of being practical.

Fusion Trees

- A ***fusion tree*** is a B-tree augmented with the preceding strategy for computing ranks quickly.
- The B-tree has order $w^{1/6}$, so its height is $O(\log_w n)$.
- Since the rank of a key in a node can be computed in time $O(1)$, the cost of a lookup, predecessor, or successor operation is $O(\log_w n)$.

Fusion Trees

- Here's the final scorecard for fusion trees.
- Notice that ***lookup*** and ***successor*** queries are unconditionally asymptotically faster than a regular balanced BST!

The Fusion Tree

- ***lookup***: $O(\log_w n)$
- ***insert***: $O(w^{2/3} \log_w n)$
- ***delete***: $O(w^{2/3} \log_w n)$
- ***max***: $O(\log_w n)$
- ***succ***: $O(\log_w n)$
- Space: $\Theta(n)$

Fusion Trees

- The mutating operations ***insert*** and ***delete*** are expensive.
- ***Idea:*** Adapt the technique from *y*-fast tries: rather than have one big fusion tree, have a bunch of smaller data structures linked together by fusion trees.

The Fusion Tree

- ***lookup***: $O(\log_w n)$
- ***insert***: $O(w^{2/3} \log_w n)$
- ***delete***: $O(w^{2/3} \log_w n)$
- ***max***: $O(\log_w n)$
- ***succ***: $O(\log_w n)$
- Space: $\Theta(n)$

Fusion Trees

- In 1996, Arne Andersson devised the ***exponential tree***, a variation on fusion trees with these indicated runtimes.
- ***Intuition:*** Instead of having a constant branching factor at each level of the tree, have the branching factor decay exponentially.
- This still keeps the tree height low, but makes the amortized cost of each operation small.

The Exponential Tree

- ***lookup***: $O(\log_w n)$
- ***insert***: $O(\log_w n + \log \log n)^*$
- ***delete***: $O(\log_w n + \log \log n)^*$
- ***max***: $O(\log_w n)$
- ***succ***: $O(\log_w n)$
- Space: $\Theta(n)$

* Amortized

A Cool Application: ***Integer Sorting***

Integer Sorting

- Suppose you're given a list of b-bit integers $x_1, x_2, \ldots, x_n$ to sort.

"Classical" techniques

- ***Heapsort*** takes time $O(n \log n)$.
- ***Base-2 radix sort*** takes time $O(nb)$.
- ***Base-n radix sort*** takes time $O(nb / \log n)$.

- A ***y-fast trie*** takes expected time $O(n \log b)$.
- An ***exponential tree*** takes time $O(n \log_b n)$.

"Modern" techniques

Integer Sorting

- These algorithms are asymptotically incomparable, since b and n are independent quantities.

y-Fast Trie Sort

$O(n \log b)$

Exponential Tree Sort

$O(n \log_b n)$

- ***Question:*** What is the crossover point?

Integer Sorting

- These algorithms are asymptotically incomparable, since b and n are independent quantities.

y-Fast Trie Sort

$O(n \log b)$

Exponential Tree Sort

$O(n \log_b n)$

- ***Question:*** What is the crossover point?

$$n \log b \quad = \quad n \log_b n$$

Integer Sorting

- These algorithms are asymptotically incomparable, since b and n are independent quantities.

y-Fast Trie Sort

$O(n \log b)$

Exponential Tree Sort

$O(n \log_b n)$

- ***Question:*** What is the crossover point?

$$n \log b = n \log_b n$$

$$\log b = \log_b n$$

Integer Sorting

- These algorithms are asymptotically incomparable, since b and n are independent quantities.

y-Fast Trie Sort

$O(n \log b)$

Exponential Tree Sort

$O(n \log_b n)$

- ***Question:*** What is the crossover point?

$$n \log b = n \log_b n$$

$$\log b = \log_b n$$

$$\log b = \frac{\log n}{\log b}$$

Integer Sorting

- These algorithms are asymptotically incomparable, since b and n are independent quantities.

***y*-Fast Trie Sort**

$O(n \log b)$

Exponential Tree Sort

$O(n \log_b n)$

- ***Question:*** What is the crossover point?

$$n \log b = n \log_b n$$

$$\log b = \log_b n$$

$$\log b = \frac{\log n}{\log b}$$

$$\log^2 b = \log n$$

Integer Sorting

- These algorithms are asymptotically incomparable, since b and n are independent quantities.

***y*-Fast Trie Sort**

$O(n \log b)$

Exponential Tree Sort

$O(n \log_b n)$

- ***Question:*** What is the crossover point?

$$n \log b = n \log_b n$$

$$\log b = \log_b n$$

$$\log b = \frac{\log n}{\log b}$$

$$\log^2 b = \log n$$

$$\log b = \sqrt{\log n}$$

Integer Sorting

- These algorithms are asymptotically incomparable, since b and n are independent quantities.

y-Fast Trie Sort

$O(n \log b)$

Exponential Tree Sort

$O(n \log_b n)$

- ***Question:*** What is the crossover point?

$$n \log b = n \log_b n$$

$$\log b = \log_b n$$

$$\log b = \frac{\log n}{\log b}$$

$$\log^2 b = \log n$$

$$\log b = \sqrt{\log n}$$

$$b = 2^{\sqrt{\log n}}$$

Integer Sorting

- These algorithms are asymptotically incomparable, since b and n are independent quantities.

y-Fast Trie Sort

$O(n \log b)$

Exponential Tree Sort

$O(n \log_b n)$

- ***Question:*** What is the crossover point?

- ***Theorem:*** There is a randomized, $O\left(n\sqrt{\log n}\right)$-time integer sorting algorithm.
- ***Proof:*** If $b \le 2^{\sqrt{\log n}}$, use exponential tree sort. Otherwise, use y-fast trie sort.

More to Explore

- In 1994, Fredman and Willard (the creators of the fusion tree) invented the ***AF-heap***, a variation on a Fibonacci heap with ***extract-min*** taking time O(log n / log log n) and used it to get a linear time algorithm for computing minimum spanning trees.
- In 1995, Andersson et al adapted the size-reduction techniques from fusion trees to develop ***signature sort***, a randomized sorting algorithm for integers. Assuming $w = \lg^{2+\varepsilon} n$, it runs in expected time O(n).
- In 1997, using the linear-time MST algorithm, Thorup developed a ***linear-time algorithm*** for undirected SSSP. (Want to learn more? Your classmates will be presenting it next Wednesday at 10:30AM!)
- In 2002, Han developed a deterministic $\mathbf{O(n \log \log n)}$-time algorithm for integer sorting that uses only linear space, and with Thorup developed a randomized $\mathbf{O}\left(n\sqrt{\log \log n}\right)$-time algorithm for integer sorting that only uses linear space.
- In 2002, Andersson and Thorup developed a deterministic, worst-case efficient integer ordered dictionary with each operation costing $\mathbf{O}\left(\sqrt{\frac{\log n}{\log \log n}}\right)$, which is provably optimal under reasonable assumptions.

Next Time

- ***Dynamic Connectivity***
  - Maintaining connectivity in a changing world.
- ***Euler Tour Trees***
  - Dynamic connectivity in forests.
- ***Dynamic Graphs***
  - A hierarchical data structure for dynamic connectivity in general undirected graphs.