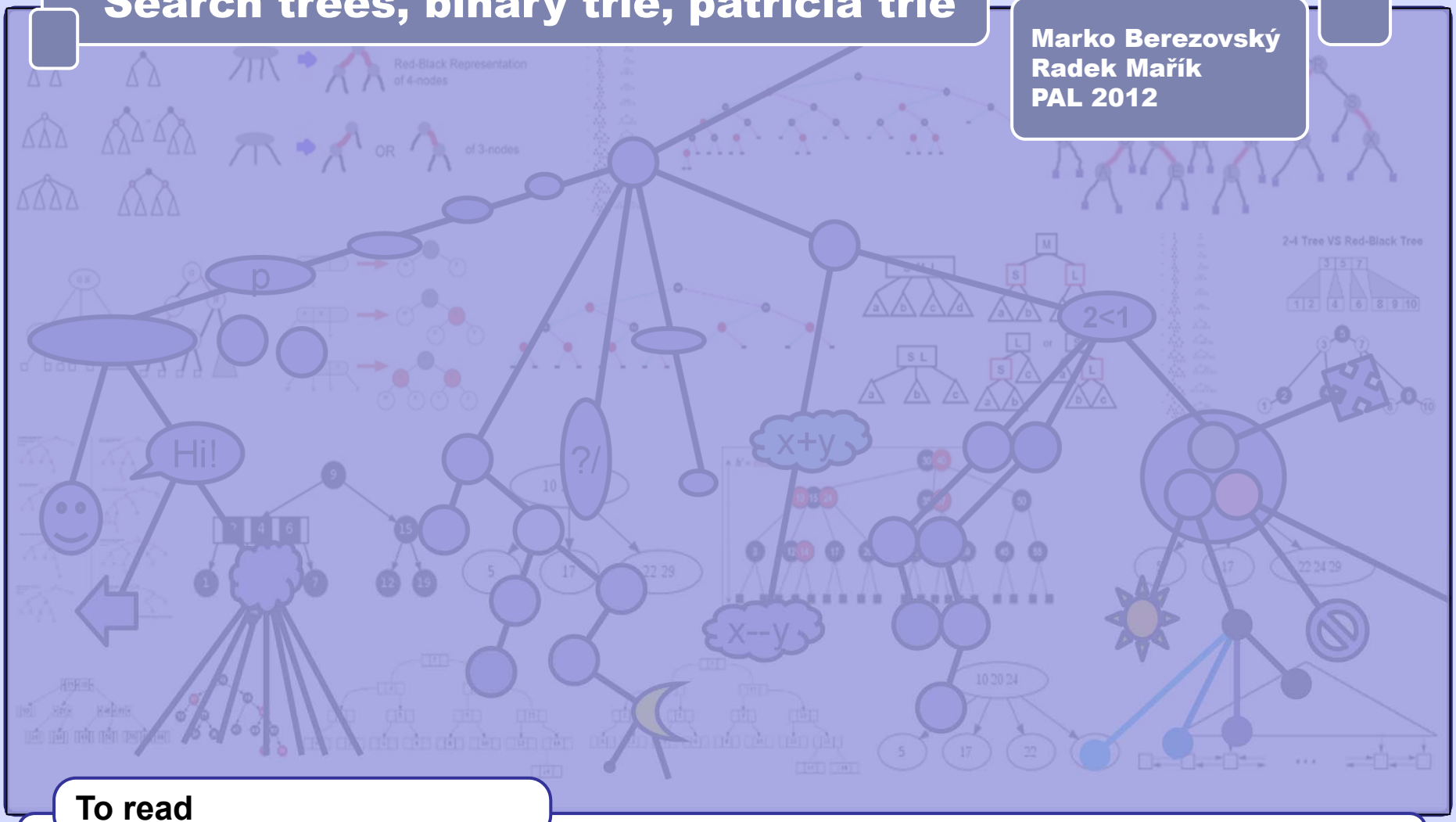
Search trees, binary trie, patricia trie

Marko Berezovský
Radek Mařík
PAL 2012



To read

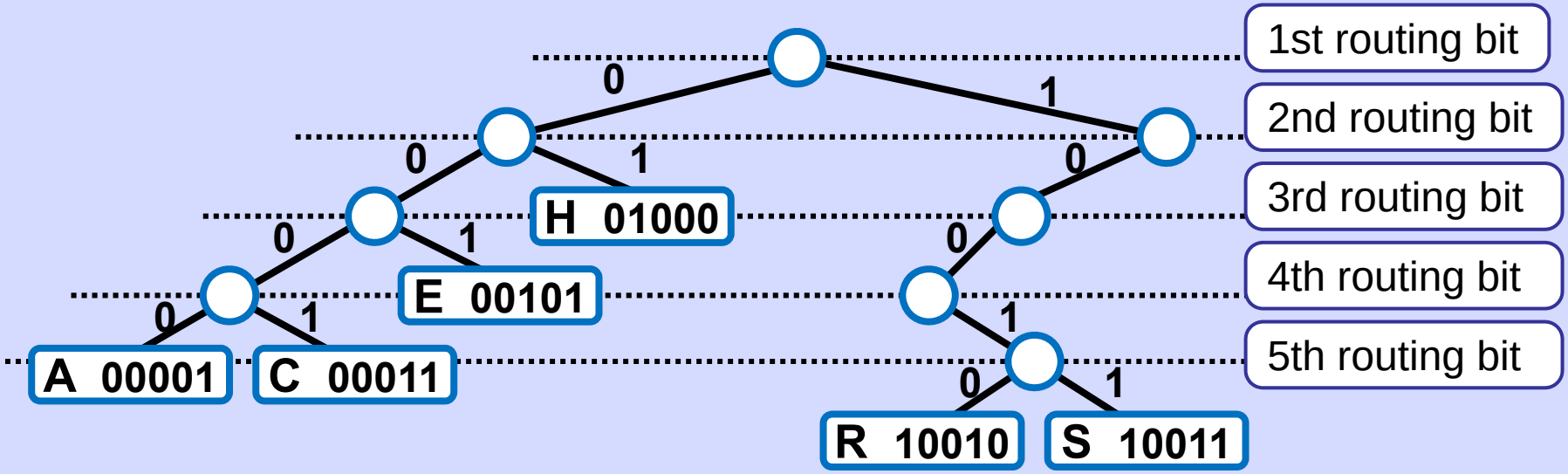
Robert Sedgewick : *Algorithms in C++, Parts 1–4: Fundamentals, Data Structure, Sorting, Searching*, Third Edition, Addison Wesley 1998, chapter 15.

See PAL webpage for references

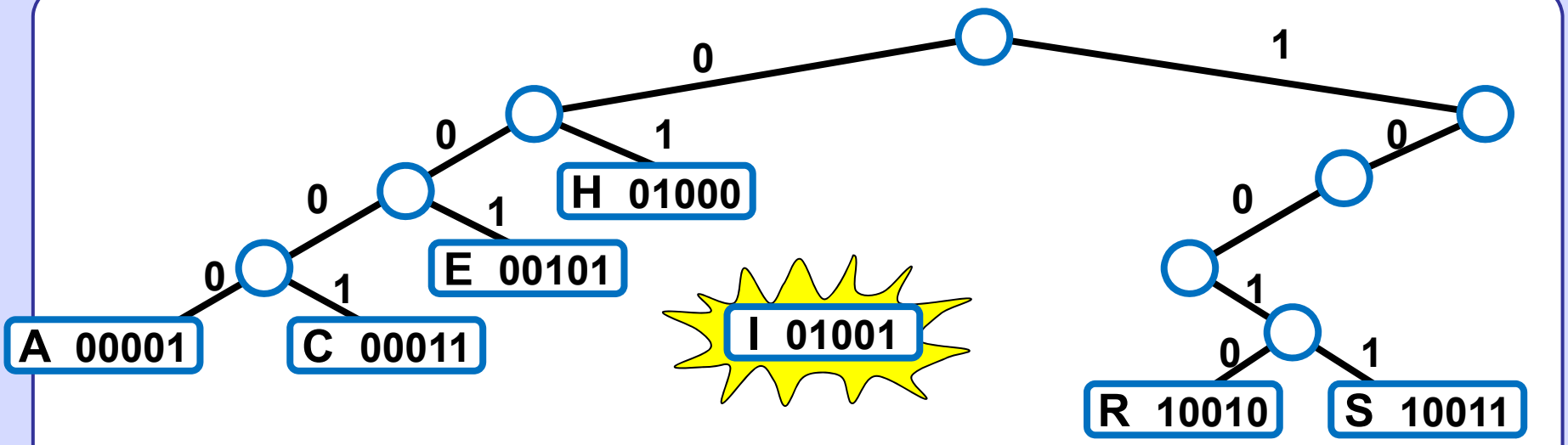
Binary trie

Keys are represented as a sequence of bits.
Keys are stored in roots only, inner nodes serve as routers only.
Inner node in depth d defines path to the leaf with key K according to the d-th bit of K .

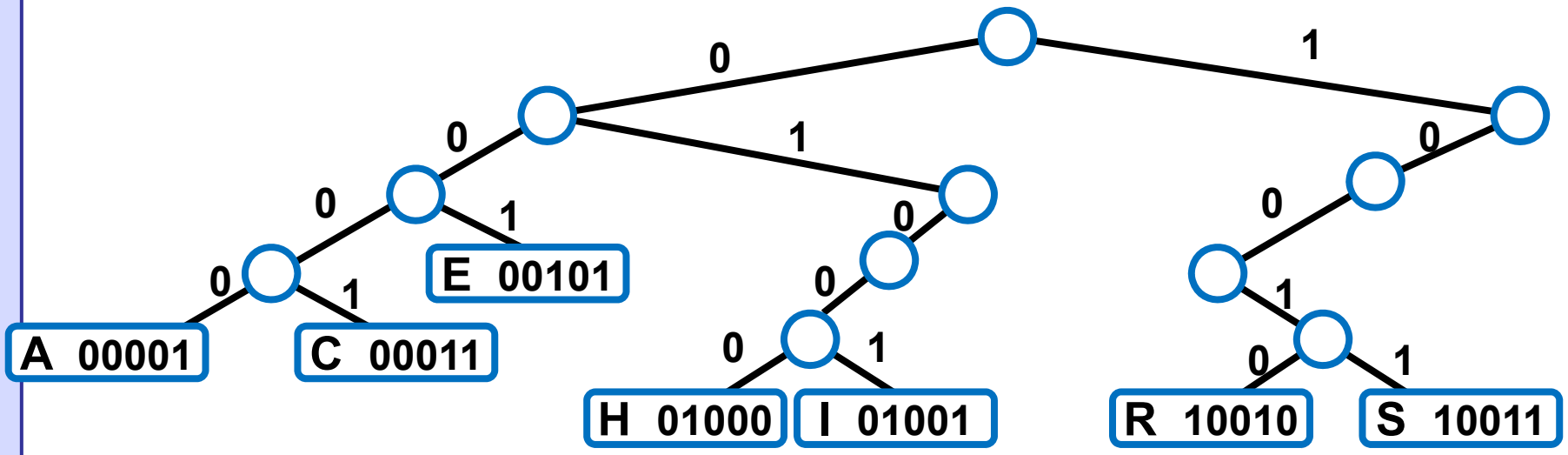
Key must not be a prefix of another key in terms of bit representation.
This can be achieved by representations having all same length in bits.



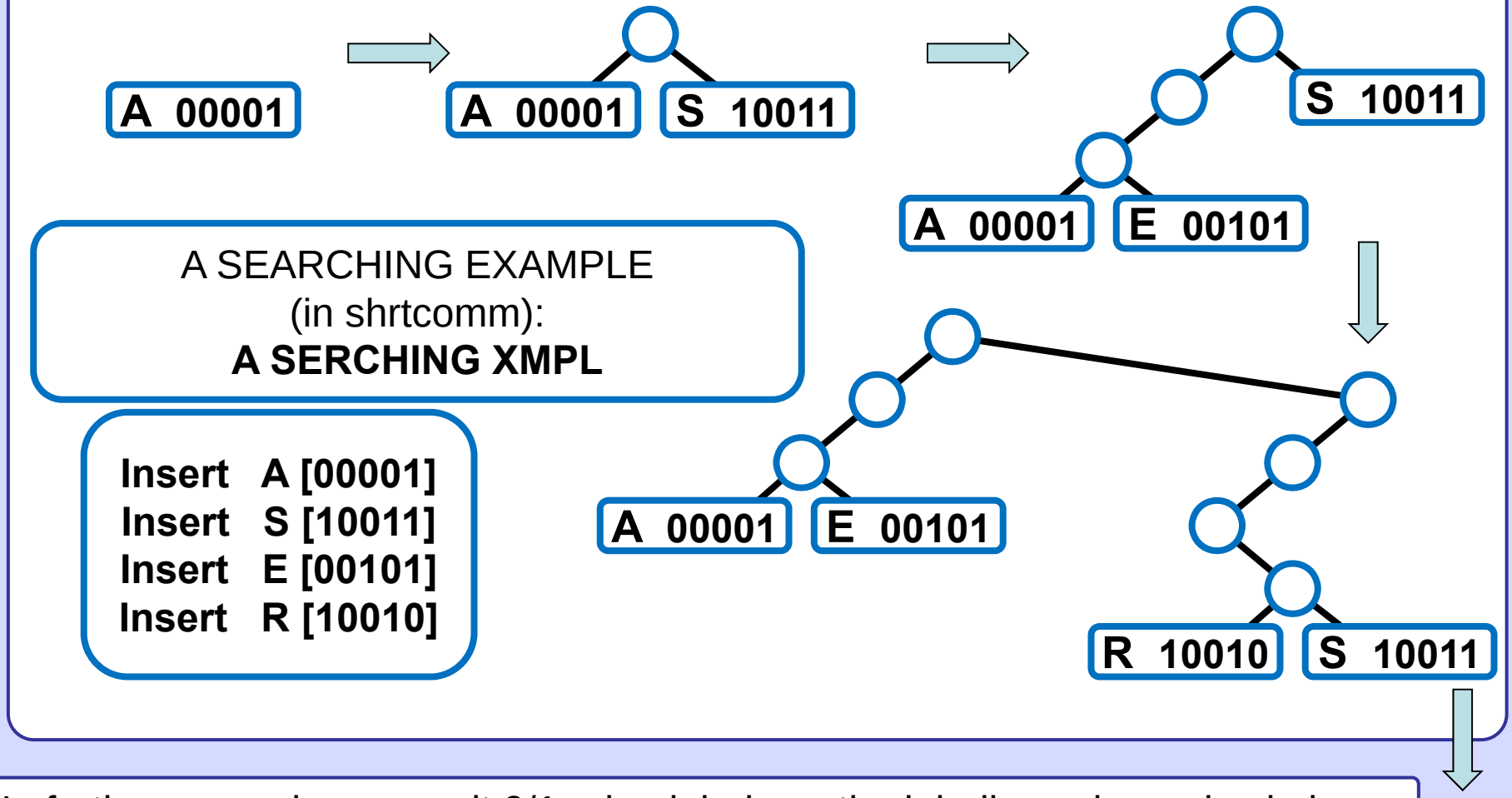
Tree height <= no of routing bits = length of bit representation of keys



Inserting single key may result in creating more internal nodes. Insert I [01001] .



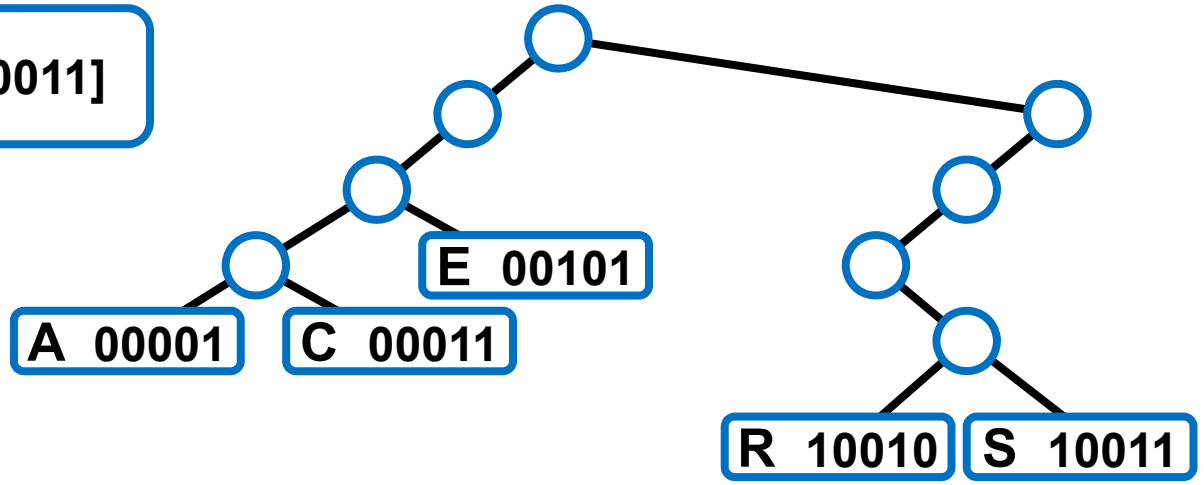
Example of trie building.



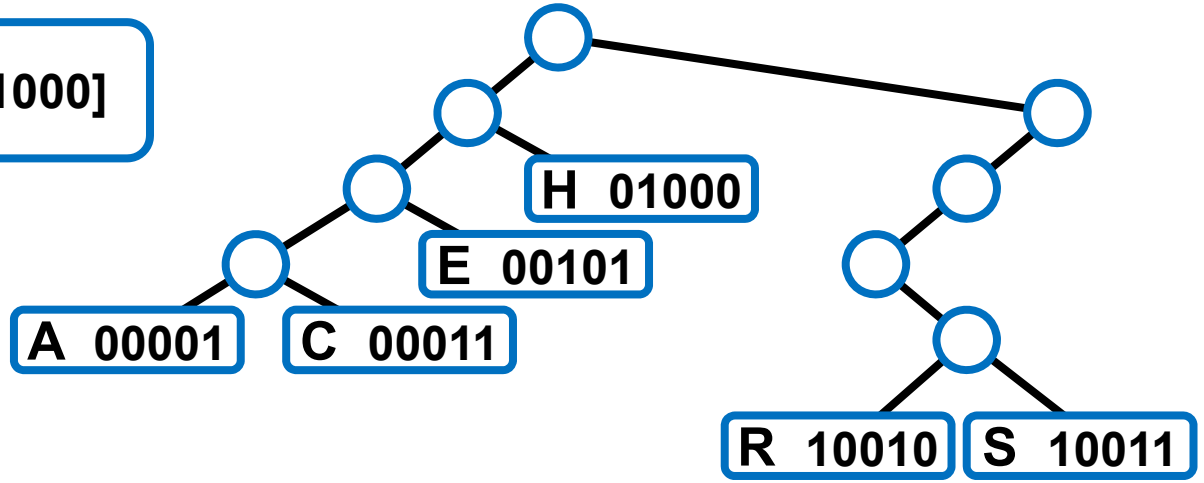
In further examples we omit 0/1 edge labels as the labeling scheme is obvious.

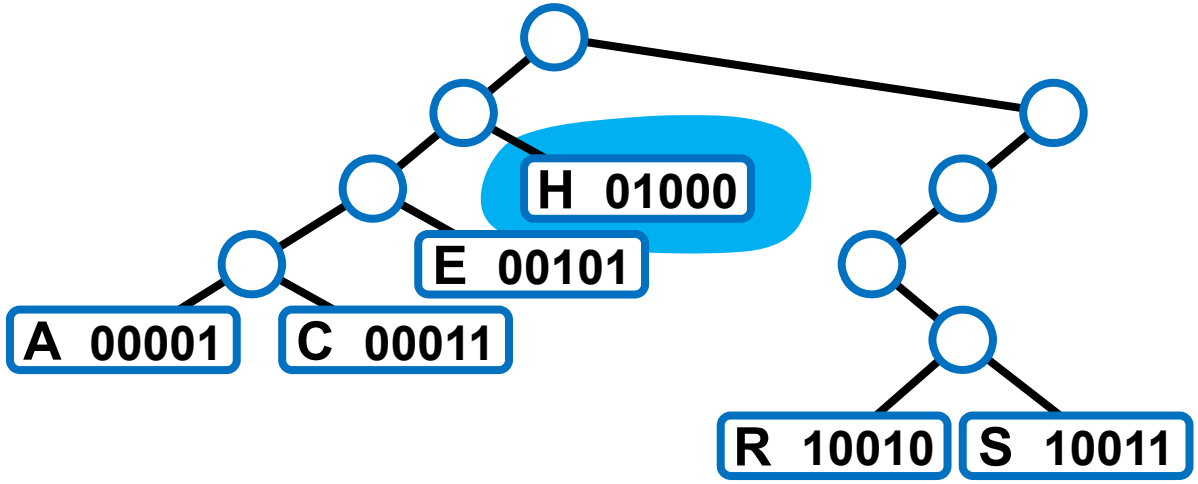
Binary trie

Insert C [00011]

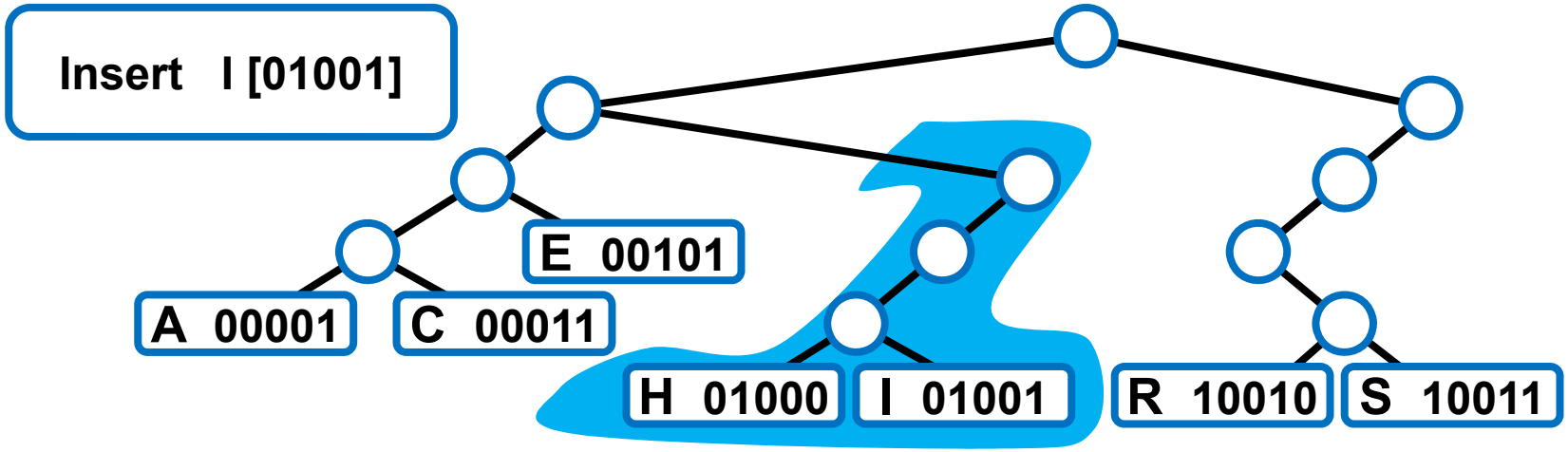


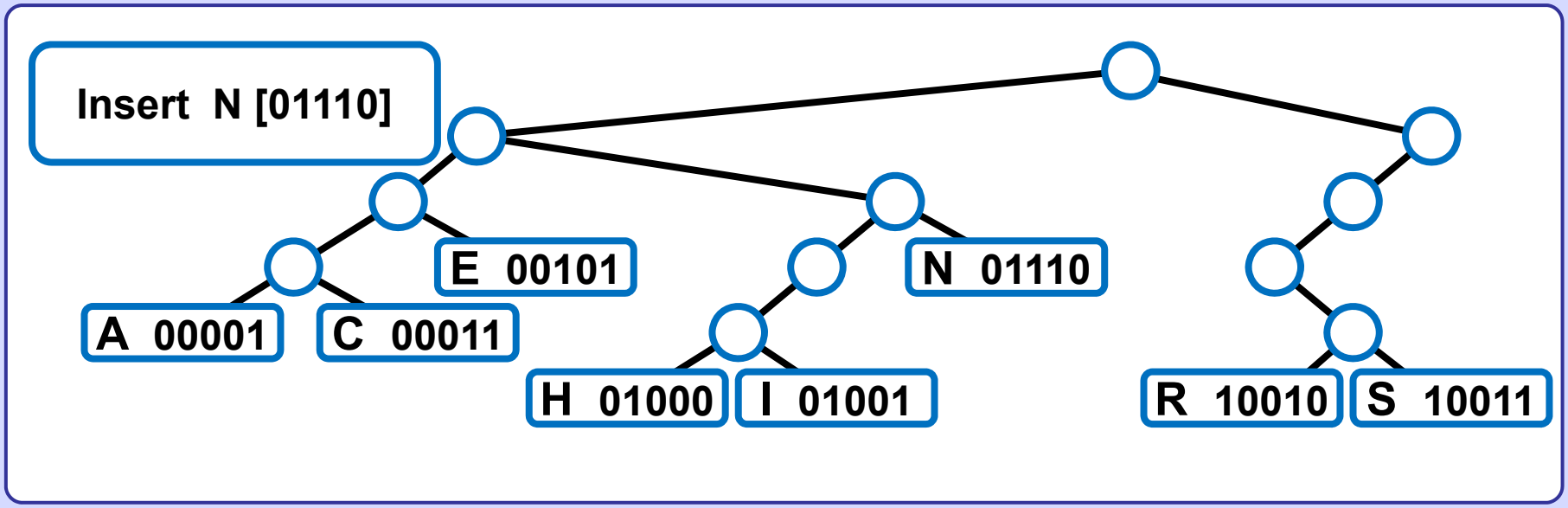
Insert H [01000]

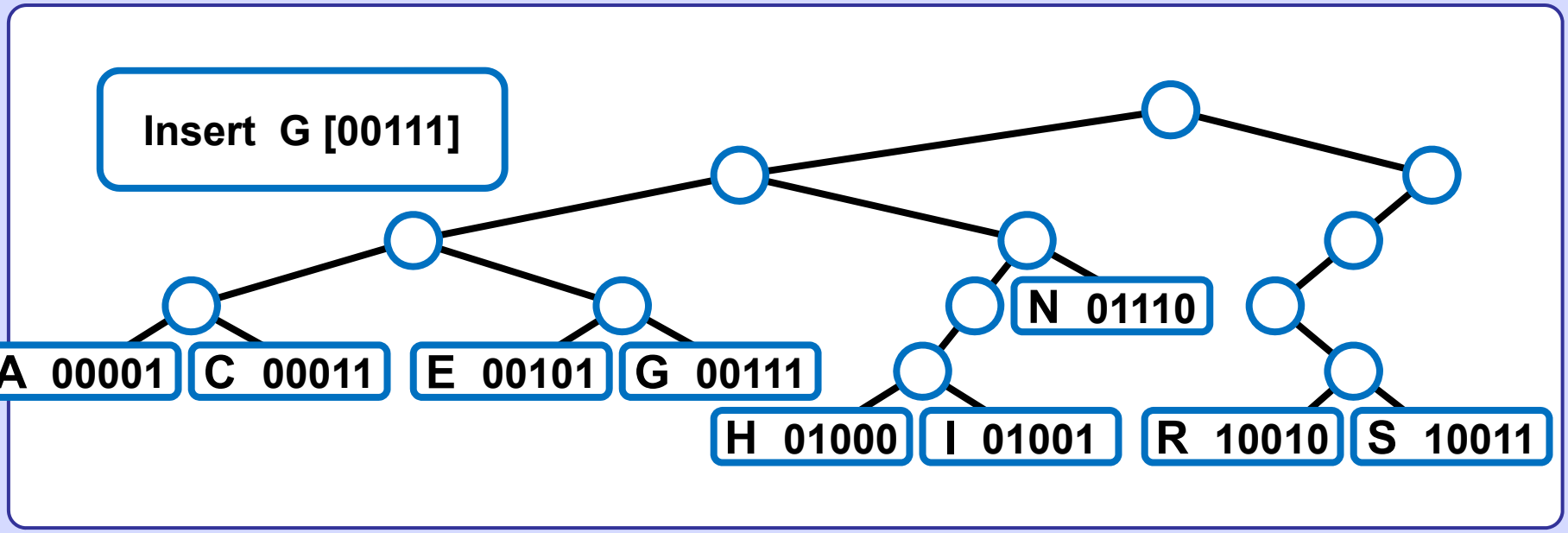


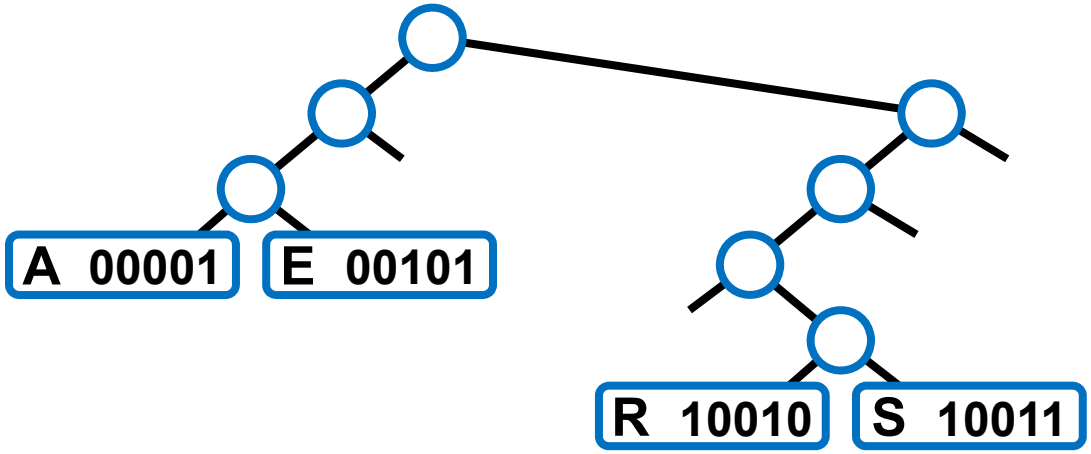


Example of multiple node splitting in one insert operation.




Insert N [01110]
E 00101
A 00001
C 00011
N 01110
H 01000
I 01001
R 10010
S 10011


Insert G [00111]
A 00001
C 00011
E 00101
G 00111
N 01110
H 01000
I 01001
R 10010
S 10011



Null pointers in internal nodes are called null links. When trie has no null link it is a complete perfectly balanced binary tree containing $2^{d+1}-1$ nodes and 2^d leaves.

The function searchR uses the bits of the key to control the branching on the way down the trie.

There are three possible outcomes:
If the search reaches a leaf (with both links null), then that is the unique node in the trie that could contain the record with key v, so we test whether that node
1. indeed contains v (search hit) or
2. some key whose leading bits match v (search miss).
If the search reaches a null link, then the parent's other link must not be null, so
3. there is some other key in the trie that differs from the search key in the corresponding bit, and we have a search miss.

The following code assumes that the keys are distinct, and (if the keys may be of different lengths) that no key is a prefix of another. The item member is not used in non-leaf nodes.

```
private:
  Item searchR(link h, Key v, int d)
    { if (h == 0) return nullItem;
      if (h->l == 0 && h->r == 0)
        { Key w = h->item.key();
          return (v == w) ? h->item : nullItem; }
      if (digit(v, d) == 0)
          return searchR(h->l, v, d+1);
      else return searchR(h->r, v, d+1);
     }

public:
  Item search(Key v)
    { return searchR(head, v, 0); }
```

To insert a new node into a trie, we search as usual, then distinguish the two cases that can occur for a search miss.

1. If the miss was not on a leaf, then we replace the null link that caused us to detect the miss with a link to a new node, as usual.

2. If the miss was on a leaf, then we use a function split to make one new internal node for each bit position where the search key and the key found agree, finishing with one internal node for the leftmost bit position where the keys differ.

The switch statement in split converts the two bits that it is testing into a number to handle the four possible cases.
If the bits are the same (case $00_2 = 0$ or $11_2 = 3$), then we continue splitting; if the bits are different (case $01_2 = 1$ or $10_2 = 2$), then we stop splitting.

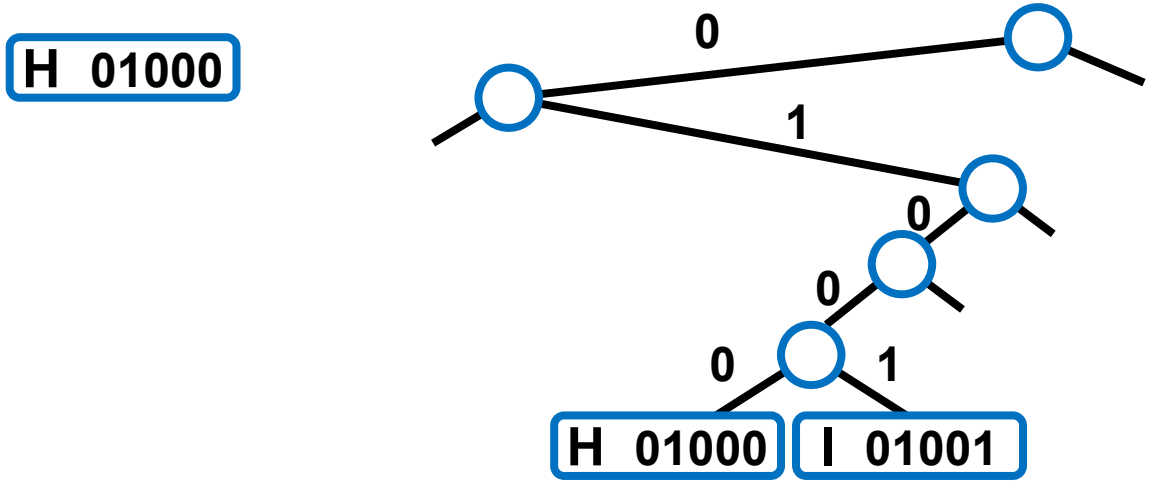
```
private:
 link split(link p, link q, int d)
   { link t = new node(nullItem); t->N = 2;
     Key v = p->item.key(); Key w = q->item.key();
     switch (digit(v, d)*2 + digit(w, d))
       { case 0: t->l = split(p, q, d+1); break;
         case 1: t->l = p; t->r = q; break;
         case 2: t->r = p; t->l = q; break;
         case 3: t->r = split(p, q, d+1); break;
       }
     return t;
   }
 void insertR(link& h, Item x, int d)
   { if (h == 0) { h = new node(x); return; }
     if (h->l == 0 && h->r == 0)
       { h = split (new node(x), h, d); return; }
     if (digit(x.key(), d) == 0)
          insertR(h->l, x, d+1);
     else insertR(h->r, x, d+1);
   }

public:  void insert(Item item) { insertR(head, item, 0); }
```

One-way branching

An annoying feature of tries, and another one that distinguishes them from the other types of search trees that we have seen, is the one-way branching required when keys have bits in common. For example, keys that differ in only the final bit always require a path whose length is equal to the key length, no matter how many keys there are in the tree, as illustrated below.
The number of internal nodes can be somewhat larger than the number of keys.

The result of inserting the keys H = 01000 and I = 01001 into an initially empty binary trie. The path length is long even with only two keys in the trie.



Binary trie shape

The trie, which part is schematically presented here, built by inserting about 200 random keys, is well-balanced, but has 44 percent more nodes than might otherwise be necessary, because of one-way branching.

Operation Delete is complementary to Insert operation.
Its algorithm/implementation is left to the reader. [Sedgewick]

The structure of a trie is independent of the key insertion order: There is a unique trie for any given set of distinct keys.

Insertion or search for a random key in a trie built from N random (distinct) bitstrings requires about lg N bit comparisons on the average. The worst-case number of bit comparisons is bounded only by the number of bits in the search key.

A trie built from N random w-bit keys has about N/ ln(2) $\approx$ 1.44 N nodes on the average.

Patricia trie

Practical Algorithm To Retrieve Information Coded in Alphanumeric

Donal R. Morrison: *PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric*, Journal of the ACM, Volume 15 Issue 4, Oct. 1968, pp 514-534.

Trie built by inserting about 200 random keys

Patricia trie built by inserting about 200 random keys

Compare how well are both trees balanced

Patricia trie

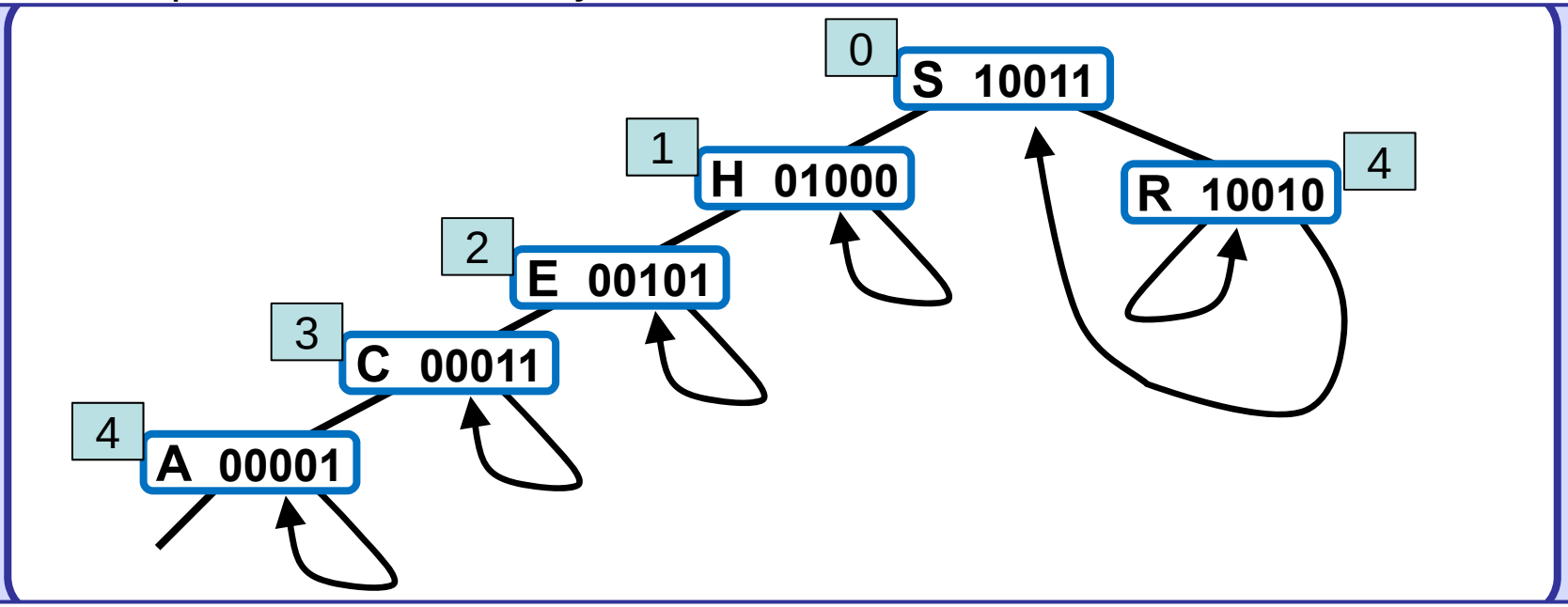
Starting with the standard trie data structure, we avoid one-way branching via a simple device: we put into each node the index of the bit to be tested to decide which path to take out of that node.
Thus, we jump directly to the bit where a significant decision is to be made, bypassing the bit comparisons at nodes where all the keys in the subtree have the same bit value.
Moreover, we avoid external nodes via another simple device: we store data in internal nodes and replace links to external nodes with links that point back upwards to the correct internal node in the trie.

These two changes allow us to represent tries with binary trees comprising nodes with a key and two links (and an additional field for the index), which we call patricia tries. With patricia tries, we store keys in nodes as with tries, and we traverse the tree according to the bits of the search key, but we do not use the keys in the nodes on the way down the tree to control the search; we merely store them there for possible later reference, when the bottom of the tree is reached.

This recursive procedure shows the records in a patricia trie in order of their keys. We imagine the items to be in (virtual) external nodes, which we can identify by testing when the bit index on the current node is not larger than the bit index on its parent. Otherwise, this program is a standard inorder traversal.

```
private:
  void showR(link h, ostream& os, int d)
    {
      if (h->bit <= d) { h->item.show(os); return; }
      showR(h->l, os, h->bit);
      showR(h->r, os, h->bit);
   }
public:
  void show(ostream& os)
    { showR(head->l, os, -1); }
```

In a successful search for R = 10010 in this sample patricia trie (top), we move right (since bit 0 is 1), then left (since bit 4 is 0), which brings us to R (the only key in the tree that begins with 1***0). On the way down the tree, we check only the key bits indicated in the numbers over the nodes (and ignore the keys in the nodes). When we first reach a link that points up the tree, we compare the search key against the key in the node pointed to by the up link, since that is the only key in the tree that could be equal to the search key.



In an unsuccessful search for I = 01001, we move left at the root (since bit 0 of the key is 0), then take the right (up) link (since bit 1 is 1) and find that H (the only key in the trie that begins with 01) is not equal to I.

The Search method differs from trie search in three ways:

- there are no explicit null links,
- we test the indicated bit in the key instead of the next bit,
- and we end with a search key comparison at the point where we follow a link up the tree.

It is easy to test whether a link points up, because the bit indices in the nodes (by definition) increase as we travel down the tree.

To search, we start at the root and proceed down the tree, using the bit index in each node to tell us which bit to examine in the search key—we go right if that bit is 1, left if it is 0. The keys in the nodes are not examined at all on the way down the tree. Eventually, an upward link is encountered: each upward link points to the unique key in the tree that has the bits that would cause a search to take that link. Thus, if the key at the node pointed to by the first upward link encountered is equal to the search key, then the search is successful; otherwise, it is unsuccessful.

The recursive function searchR returns the unique node that could contain the record with key v. It travels down the trie, using the bits of the tree to control the search, but tests only 1 bit per node encountered—the one indicated in the bit field. It terminates the search when it encounters an external link, one which points up the tree. The public function search calls searchR, then tests the key in that node to determine whether the search is a hit or a miss.

```
private:
  Item searchR(link h, Key v, int d) {
      if (h->bit <= d) return h->item; // upward link, stop
      if (digit(v, h->bit) == 0)
          return searchR(h->l, v, h->bit);
       else  return searchR(h->r, v, h->bit);
    }
public:
  Item search(Key v)
    { Item t = searchR(head, v, -1);
      return (v == t.key()) ? t : nullItem;
    }
```

Patricia trie Search

Trie search

```
private:   Item searchR(link h, Key v, int d)
    { if (h == 0) return nullItem;
      if (h->l == 0 && h->r == 0)
        { Key w = h->item.key();
          return (v == w) ? h->item : nullItem; }
      if (digit(v, d) == 0)
          return searchR(h->l, v, d+1);
      else return searchR(h->r, v, d+1);  }
public:  Item search(Key v)
    { return searchR(head, v, 0); }
```

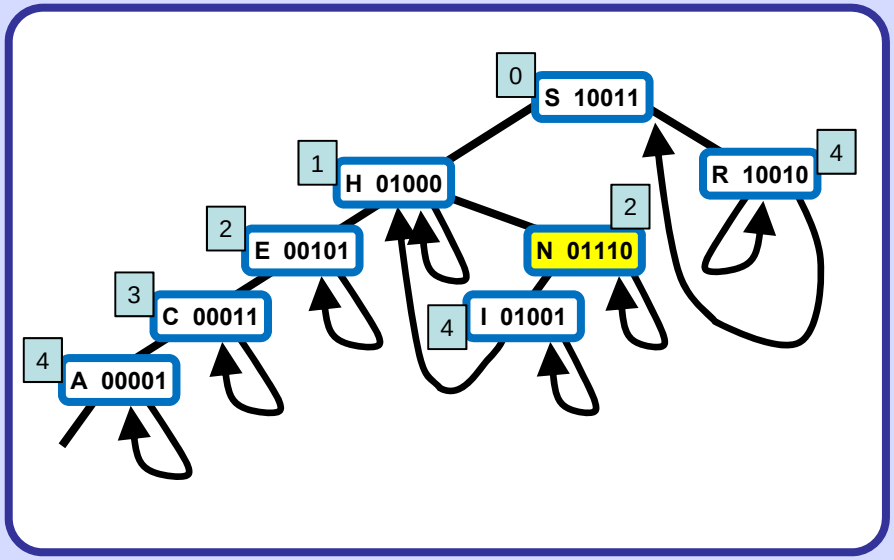
Patricia search

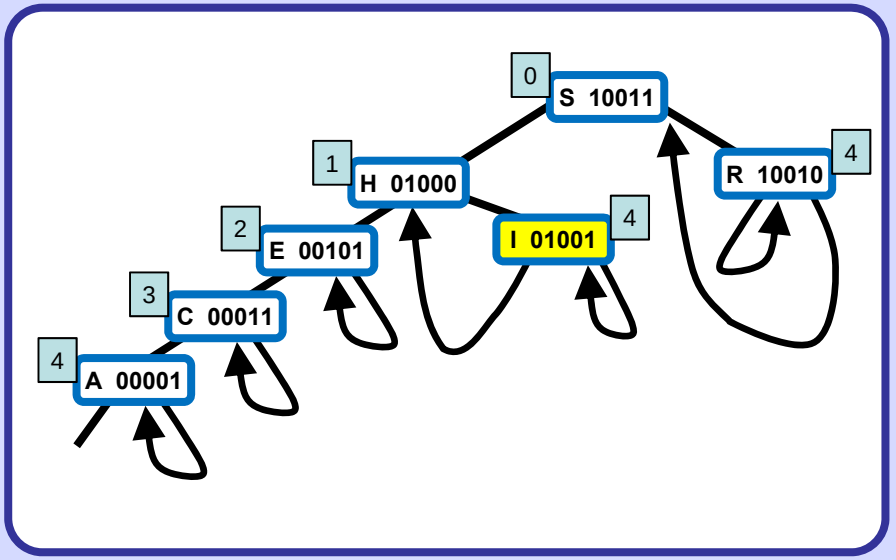
```
private:  Item searchR(link h, Key v, int d) {
     if (h->bit <= d) return h->item;
     if (digit(v, h->bit) == 0)
         return searchR(h->l, v, h->bit);
    else  return searchR(h->r, v, h->bit);  }
public:  Item search(Key v)
    { Item t = searchR(head, v, -1);
      return (v == t.key()) ? t : nullItem;  }
```

Patricia trie Insert

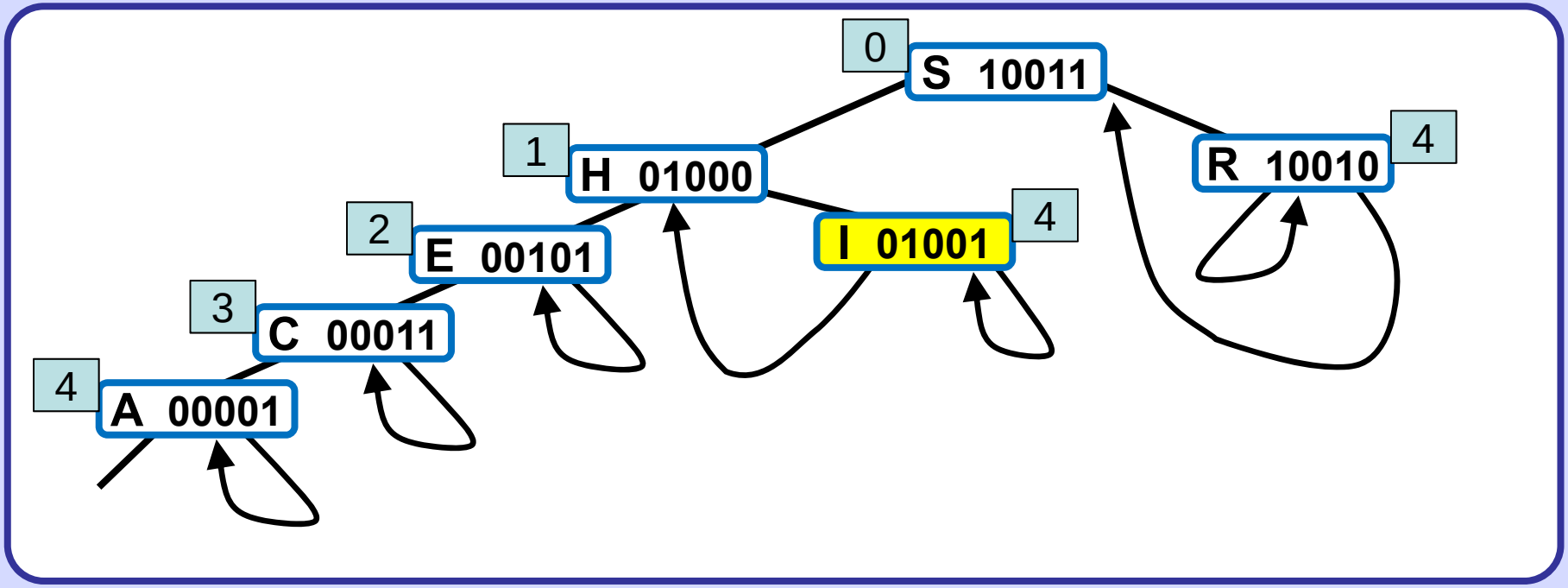
To insert a key into a patricia trie, we begin with a search. The function searchR gets us to a unique key in the tree that must be distinguished from the key to be inserted. We determine the leftmost bit position at which this key and the search key differ, then use the recursive function insertR to travel down the tree and to insert a new node containing v at that point.

In insertR, there are two cases.
1. The new node could replace an internal link (if the search key differs from the key found in a bit position that was skipped), or
2. an external link (if the bit that distinguishes the search key from the found key was not needed to distinguish the found key from all the other keys in the trie).

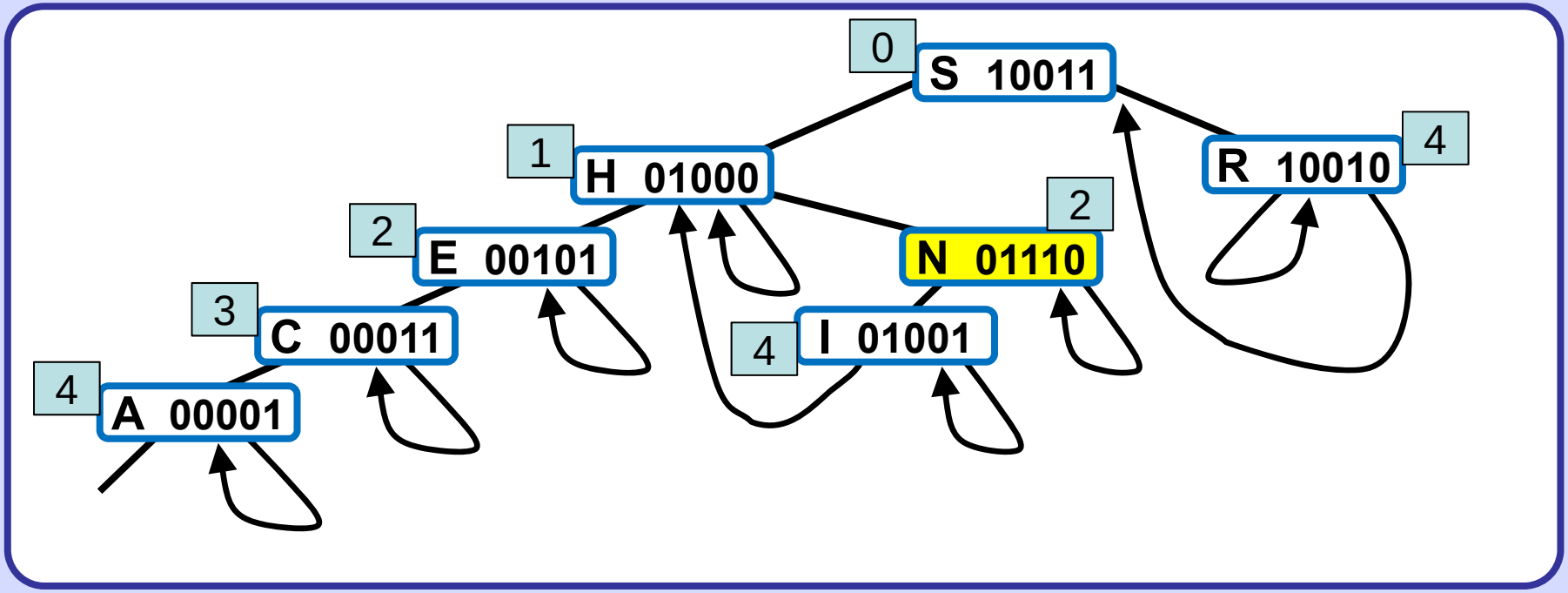




To insert I into the sample patricia trie, we add a new node to check bit 4, since H = 01000 and I = 01001 differ in only that bit (top). On a subsequent search in the trie that comes to the new node, we want to check H (left link) if bit 4 of the search key is 0; if the bit is 1 (right link), the key to check is I.



To insert N = 01110, we add a new node in between H and I to check bit 2, since that bit distinguishes N from H and I.



The implementation of insertion for patricia tries mirrors the two cases that arise in insertion for tries, as illustrated in previous example.
As usual, we gain information on where a new key belongs from a search miss.

For tries, the miss can occur
either because of a null link
or because of a key mismatch at a leaf.

For patricia tries, we need to do more work to decide which type of insertion is needed, because we skipped the bits corresponding to one-way branching during the search.
A patricia-trie search always ends with a key comparison, and this key carries the information that we need.

We find the leftmost bit position where the search key and the key that terminated the search differ, then search through the trie again, comparing that bit position against the bit positions in the nodes on the search path.
If we come to a node that specifies a bit position higher than the bit position that distinguishes the key sought and the key found, then we know that we skipped a bit in the patricia-trie search that would have led to a null link in the corresponding trie search, so we add a new node for testing that bit.
If we never come to a node that specifies a bit position higher than the one that distinguishes the key sought and the key found, then the patricia-trie search corresponds to a trie search ending in a leaf, and we add a new node that distinguishes the search key from the key that terminated the search.

We always add just one node, which references the leftmost bit that distinguishes the keys, where standard trie insertion might add multiple nodes with one-way branching before reaching that bit. That new node, besides providing the bit-discrimination that we need, will also be the node that we use to store the new item.

```
public:
  void insert(Item x)
    { Key v = x.key(); int i;
      Key w = searchR(head->l, v, -1).key();
      if (v == w) return; // no duplicates
      for (i = 0; digit(v, i) == digit(w, i); i++) ;
      head->l = insertR(head->l, x, i, head);
    }

  ST( int maxN)
    { head = new node(nullItem);
      head->l = head->r = head; }
```

```
private:
  link insertR(link h, Item x, int d, link p)
    { Key v = x.key();
      if ((h->bit >= d) || (h->bit <= p->bit))
        {
          link t = new node(x); t->bit = d;
          t->l = (digit(v, t->bit) ? h : t);
          t->r = (digit(v, t->bit) ? t : h);
          return t;
        }
      if (digit(v, h->bit) == 0)
            h->l = insertR(h->l, x, d, h);
      else  h->r = insertR(h->r, x, d, h);
      return h;
   }
```

The search cost in a standard trie typically does depend on the length of the keys—the first bit position that differs in two given keys could be arbitrarily far into the key.

All the comparison-based search methods that we have considered so far also depend on the key length—if two keys differ in only their rightmost bit, then comparing them requires time proportional to their length.

Hashing methods always require time proportional to the key length for a search, to compute the hash function. But patricia immediately takes us to the bits that matter, and typically involves testing less than lg N of them. This effect makes patricia (or trie search with one-way branching removed) the search method of choice when the search keys are long.

For example, suppose that we have to search among millions of 1000-bit keys. Then patricia would require accessing only about 20 bytes of the search key for the search, plus one 125-byte equality comparison, whereas hashing would require accessing all 125 bytes of the search key to compute the hash function, plus a few equality comparisons, and comparison-based methods would require 20 to 30 full key comparisons.