Tries & Binary Search Trees

Applied Algorithms

Tries

- A trie is a tree-based data structure for storing strings in order to support fast pattern matching.
- The main application domain is information retrieval.
- Video: https://www.youtube.com/watch?v=-urNrIAQnNo

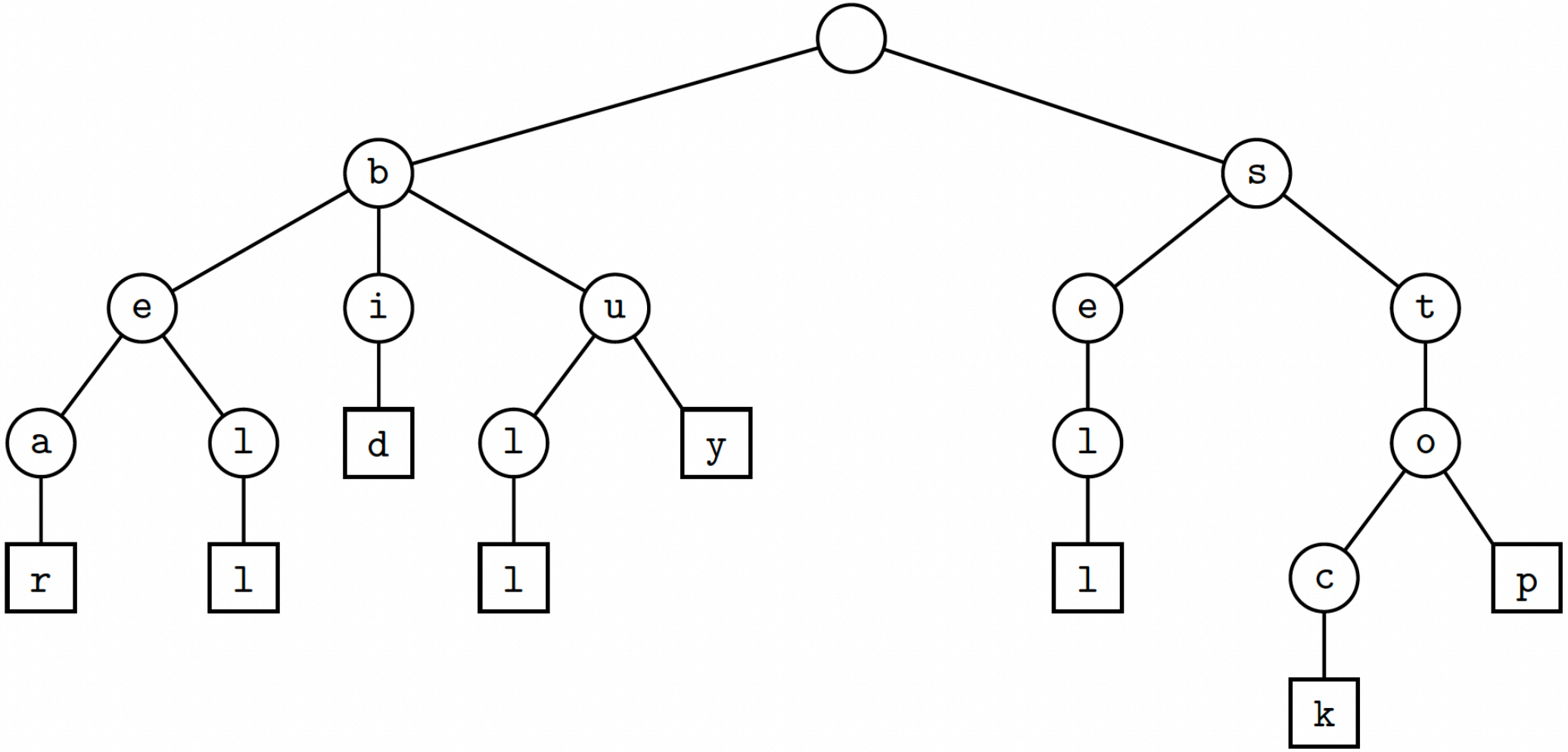
Standard Tries

- Σ : an alphabet
- S: a set of s strings. There are none string whose prefix of another string in S.
- The properties of a standard trie for S (an ordered tree) as follows:
  - Each node of the tree is labeled with a character in Σ (except the root.)
  - The children of an internal node of the tree have distinct labels.
  - Tree has s leaves. The path from the root to a leaf yields the string of S.

Standard Tries

- The tree has at most n+1 nodes
- Each internal node has at most $|\Sigma|$ children.
- The height of the tree is equal to the longest string in S.

Example: Standard Tries



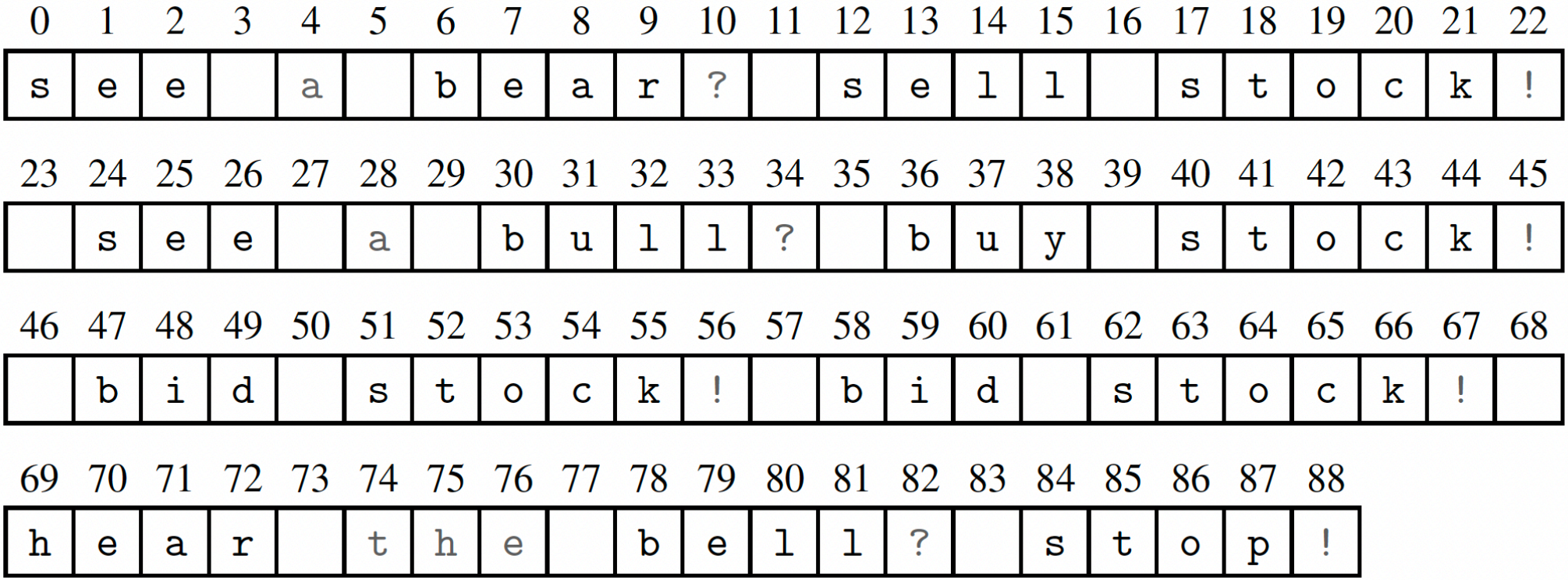
Standard trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}.

Standard Tries

- The search for a string of length m is in O(m |Σ|) time—visiting at most m+1 nodes and spending O(|Σ|) time at each node. “But, typically expect a search for a string of length m to run in O(m) time”.
- The running time to insert: worst-case O(m|Σ|), expected O(m) if using secondary hash tables at each node.
- Constructing the entire trie takes expected O(n) time (n: the total length of the strings of S).

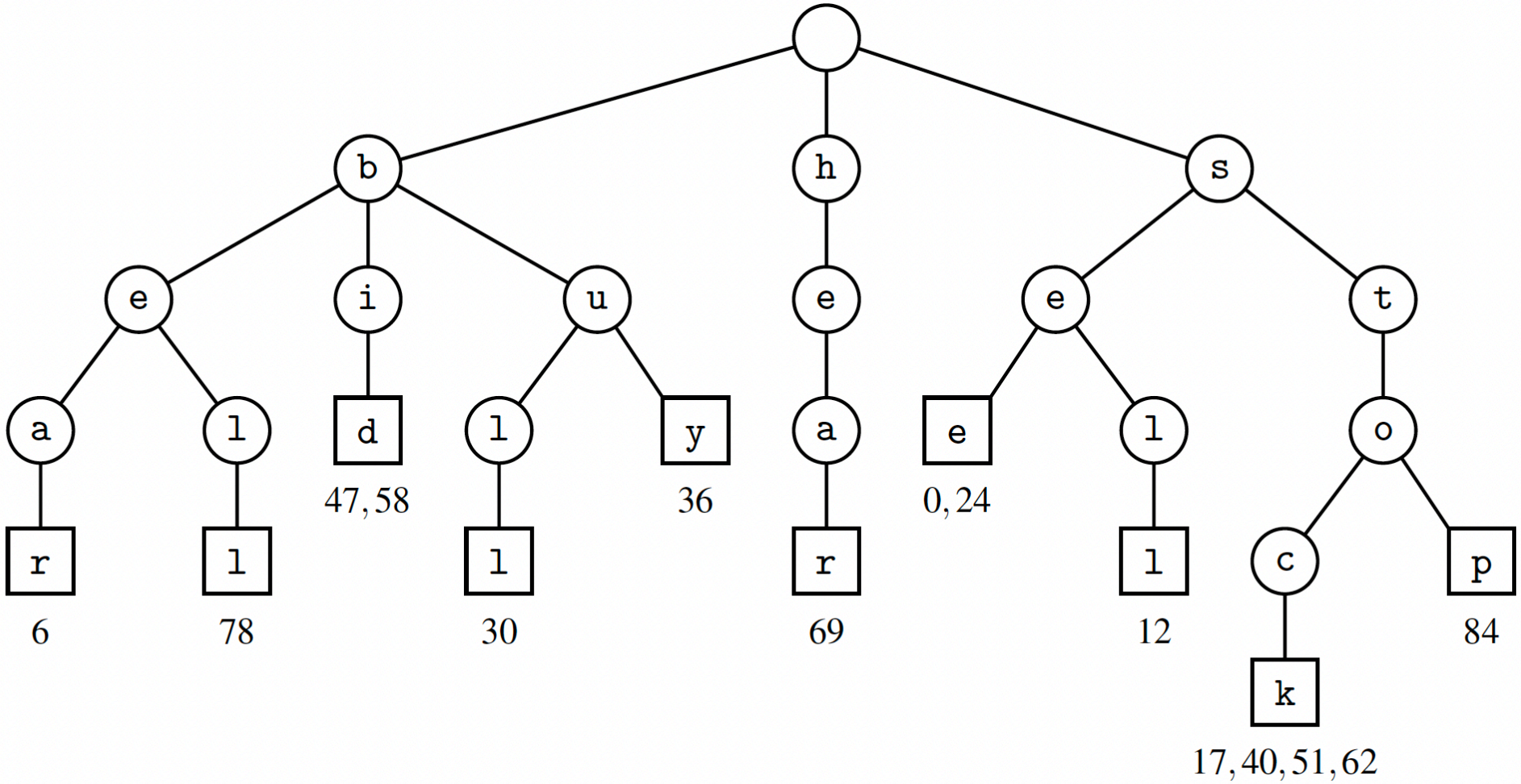
Example: Standard Tries

- Word matching with a standard trie (exclude stop words).



Example: Standard Tries

- Word matching with a standard trie:

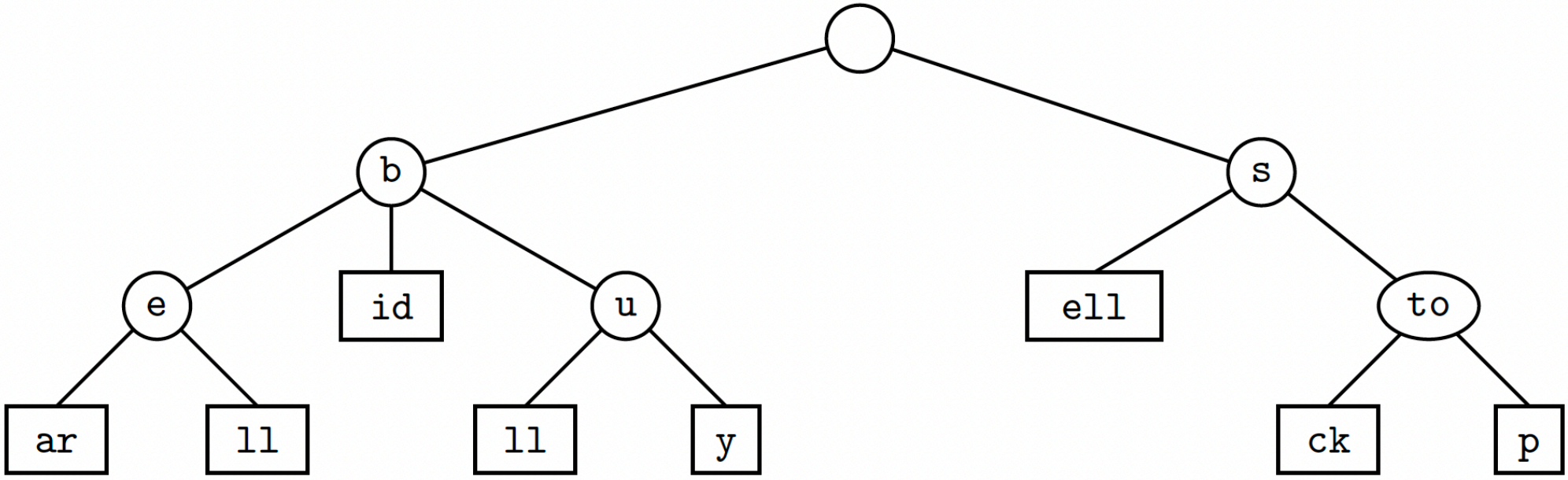


Compressed Tries

- There is a potential space inefficiency in the standard trie—a lot of nodes that have only one child.
- A compressed trie stores s strings from an alphabet of size d and has the following properties:
  - Every internal node has at least two children and most d children.
  - Tree has s leaves nodes.
  - The number of nodes of the tree is O(s).

Example: Compressed trie

- Compressed trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}.



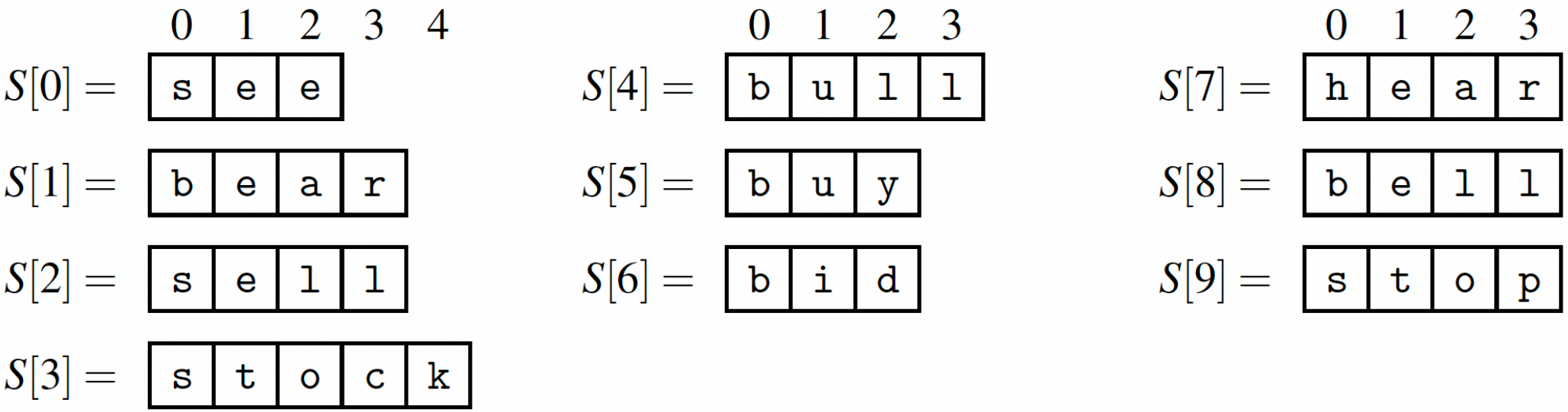
Compressed Tries

- “A compressed trie is advantageous only when it is used as an auxiliary index structure over a collection of strings already stored in a primary structure, and is not required to actually store all the characters of the strings in the collection.”

- Video: https://www.youtube.com/watch?v=lgc3_xaj4kE&t=0s

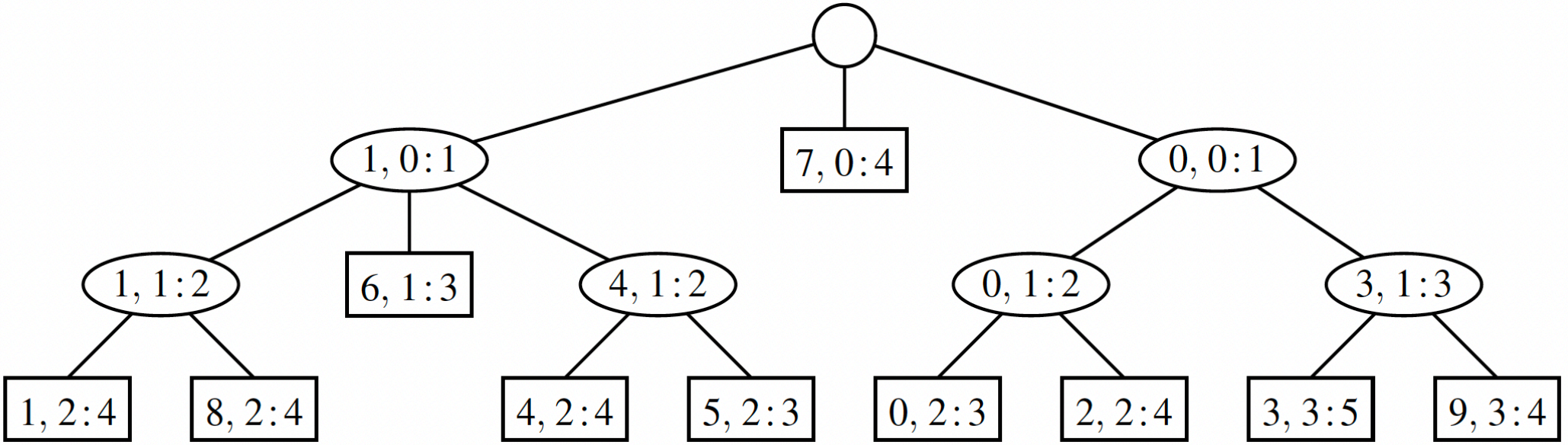
Example: compressed trie

- Collection S of strings stored in an array:



Example: compressed trie

- Compact representation:



Example: compressed trie

- The total space from O(n) for the standard trie to O(s) for the compressed trie (n: the total length of the strings in S, s: the number of strings in S).

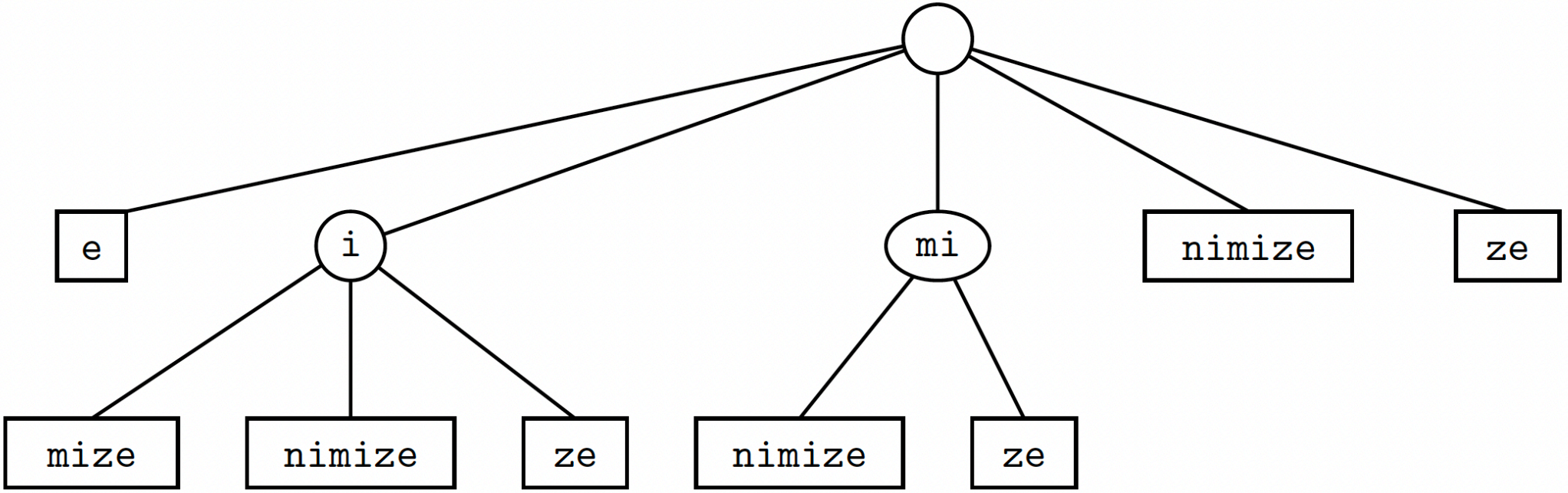
- “Searching in a compressed trie is not necessarily faster than in a standard tree, since there is still need to compare every character of the desired pattern with the potentially multi-character labels while traversing paths in the trie.”

Suffix Tries

- S includes all the suffixes of a string X. Such a trie is called the suffix trie
- The compact representation of a suffix trie uses O(n) space ($|X| = n$)
- The construction takes $O(|\Sigma|n^2)$ time. However, the compact suffix trie can be constructed in O(n) time.
- Video: https://www.youtube.com/watch?v=ddPQKSk5W6Y

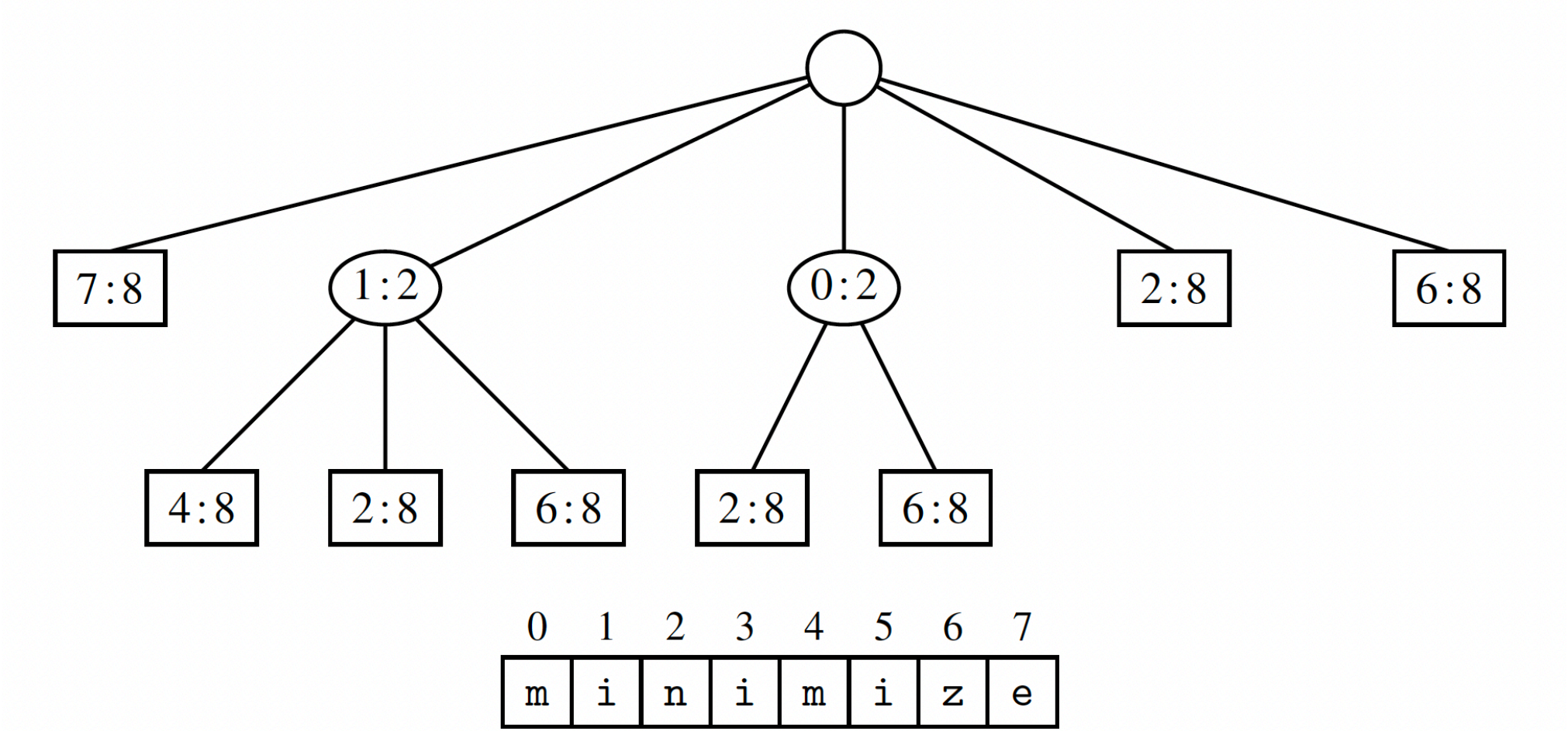
Example: Suffix Tries

- String X = “minimize”.



Example: Suffix Tries

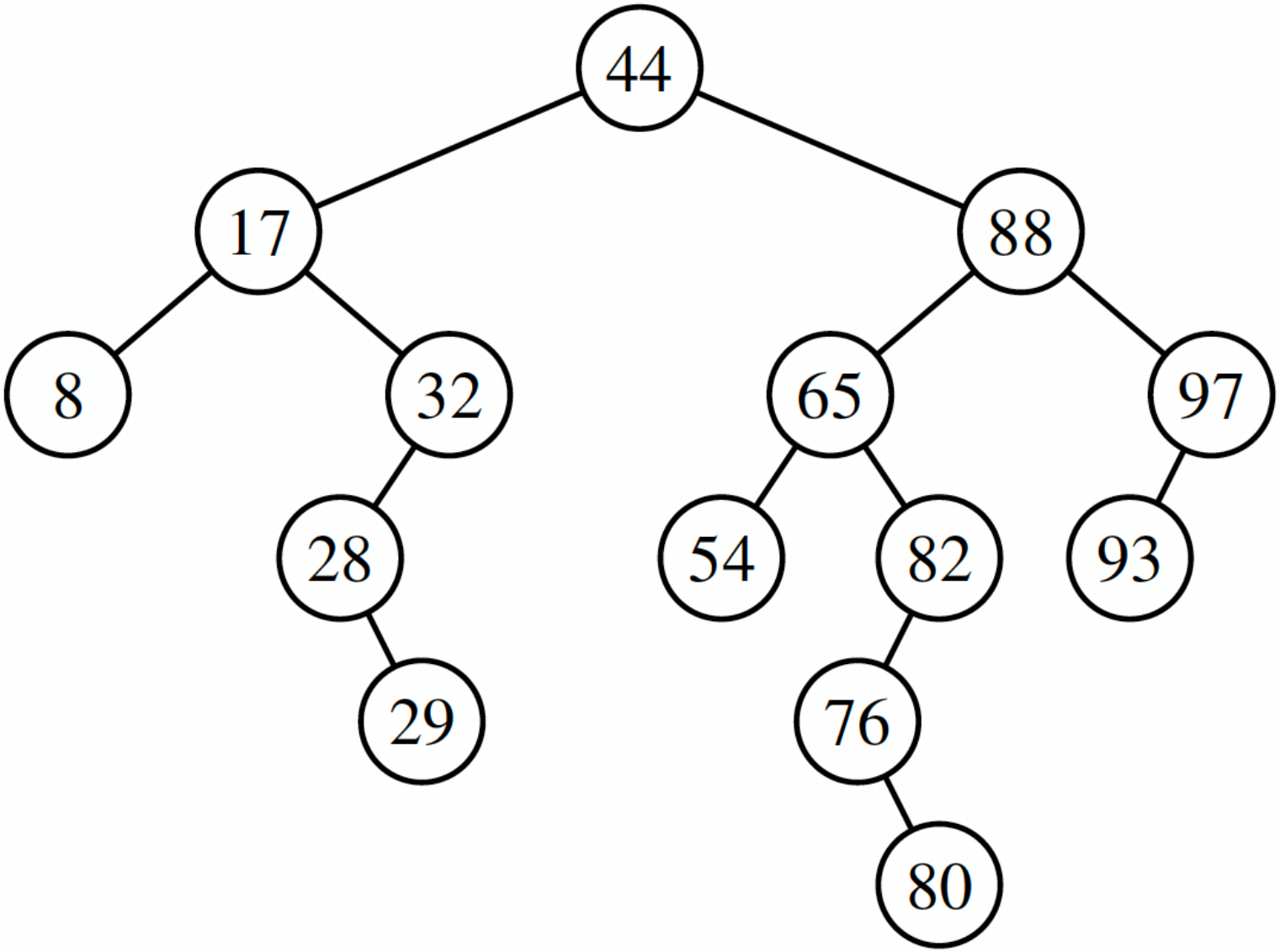
- Compact representation:



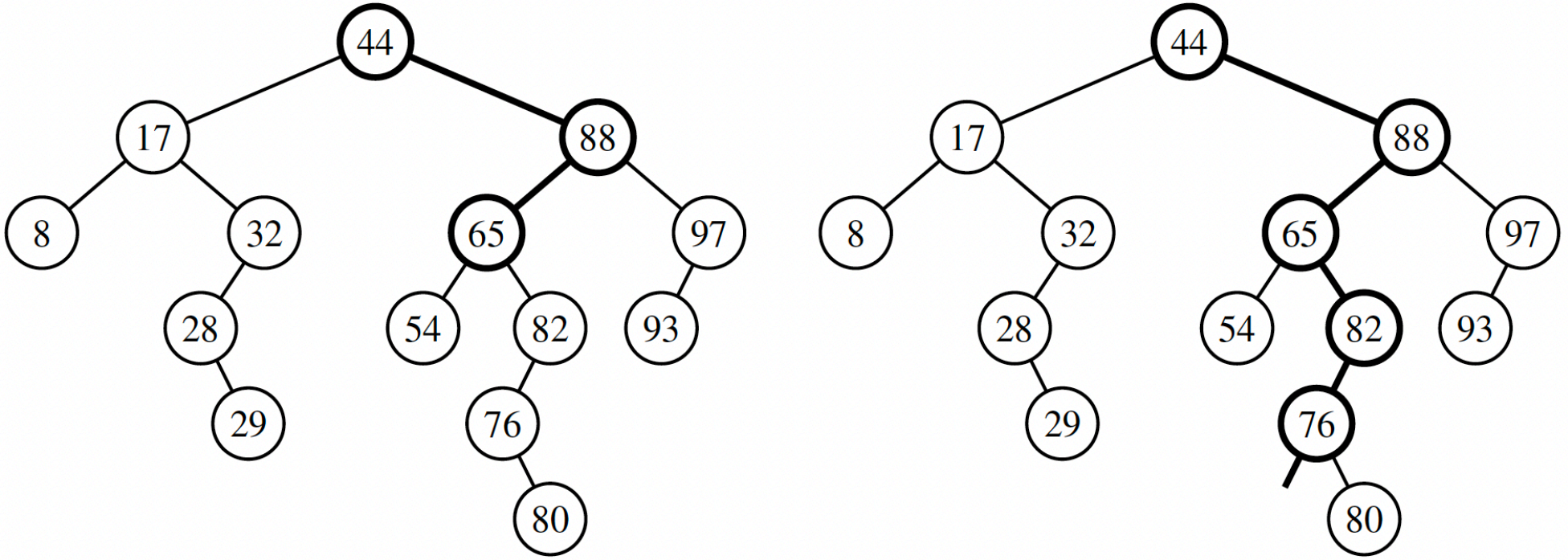
Binary Search Trees

- “A binary search tree is a binary tree and each position p storing a key-value pair (k,v) such that:
  - Keys in the left subtree of p are less than k.
  - Keys in the right subtree of p are greater than k.”
  - Video: https://www.youtube.com/watch?v=mtvbVLK5xDQ

Binary Search Trees: Example



Searches



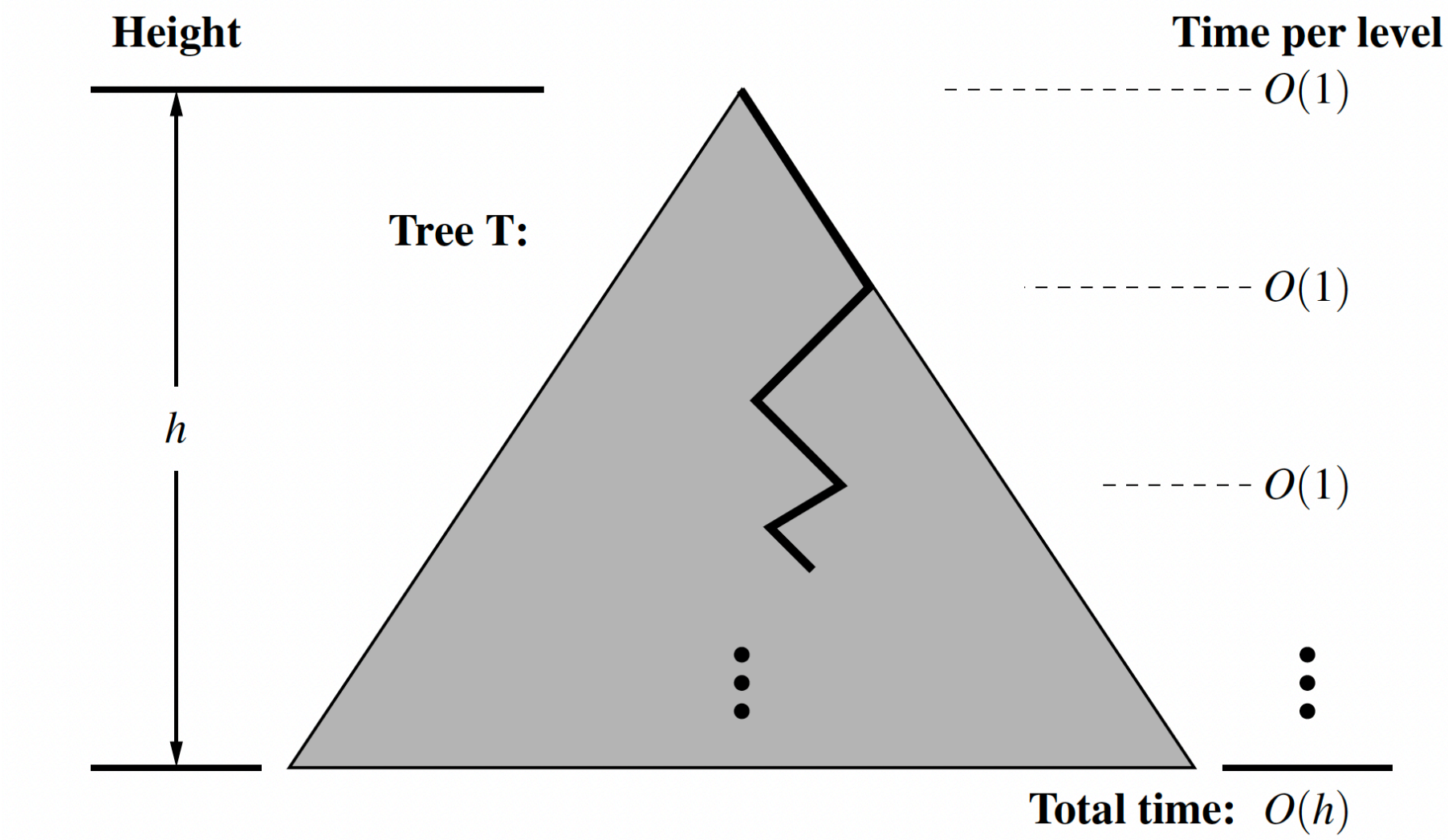
Search: key 65

Searches

```
Algorithm TreeSearch(T, p, k):
    if k == p.key() then
        return p                                          {successful search}
    else if k < p.key() and T.left(p) is not None then
        return TreeSearch(T, T.left(p), k)                {recur on left subtree}
    else if k > p.key() and T.right(p) is not None then
        return TreeSearch(T, T.right(p), k)               {recur on right subtree}
    return p                                              {unsuccessful search}
```

Recursive search in a binary search tree.

Searches



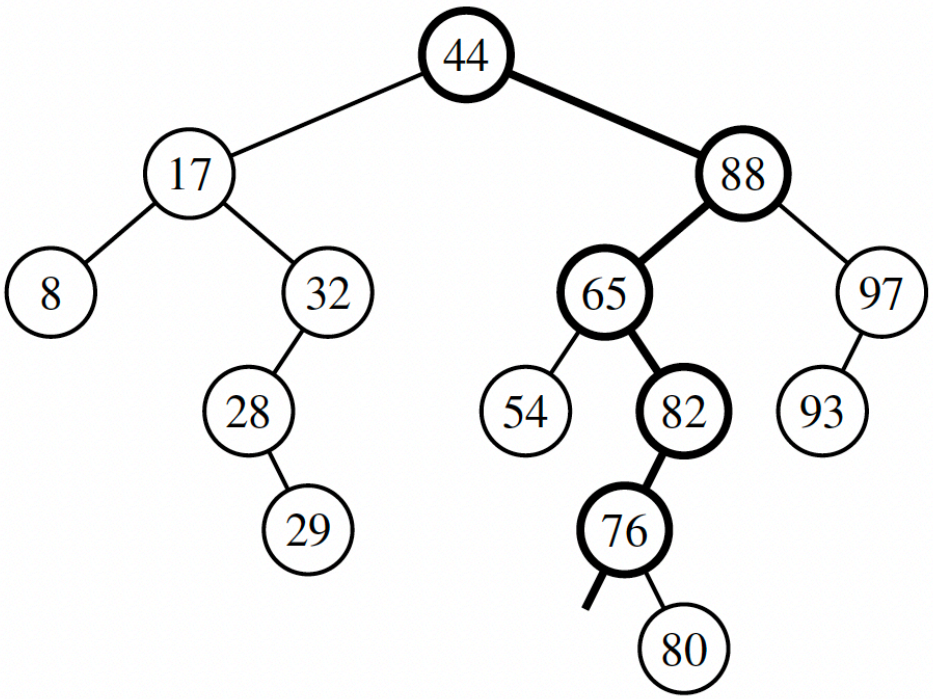
Insertion

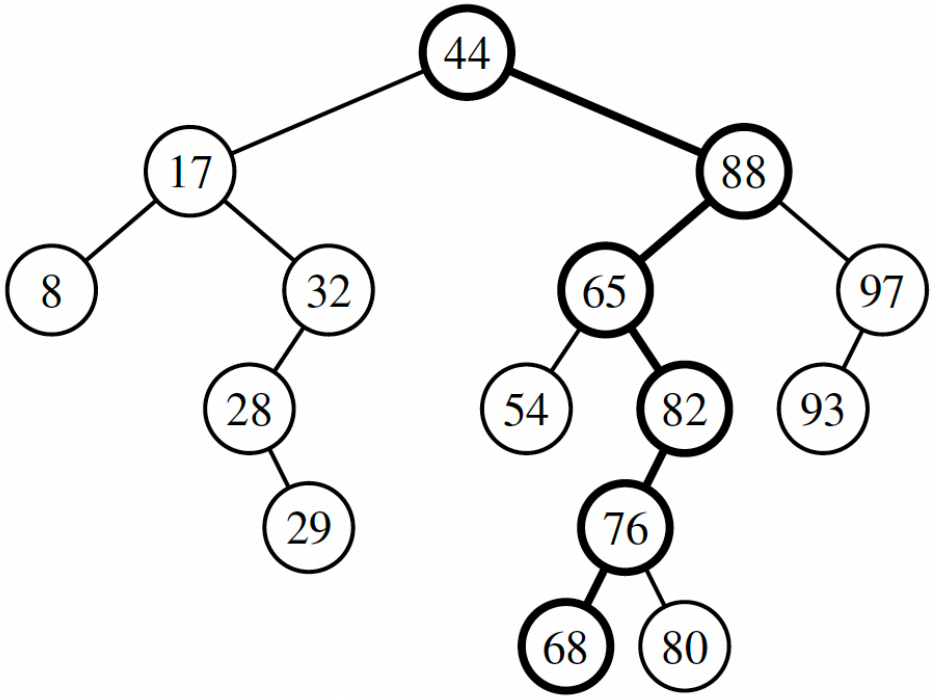
Algorithm TreeInsert(T, k, v):

Input: A search key k to be associated with value v

p = TreeSearch(T, T.root(), k)
if k == p.key() **then**
    Set p's value to v
else if k < p.key() **then**
    add node with item (k,v) as left child of p
else
    add node with item (k,v) as right child of p

Insertion: Example



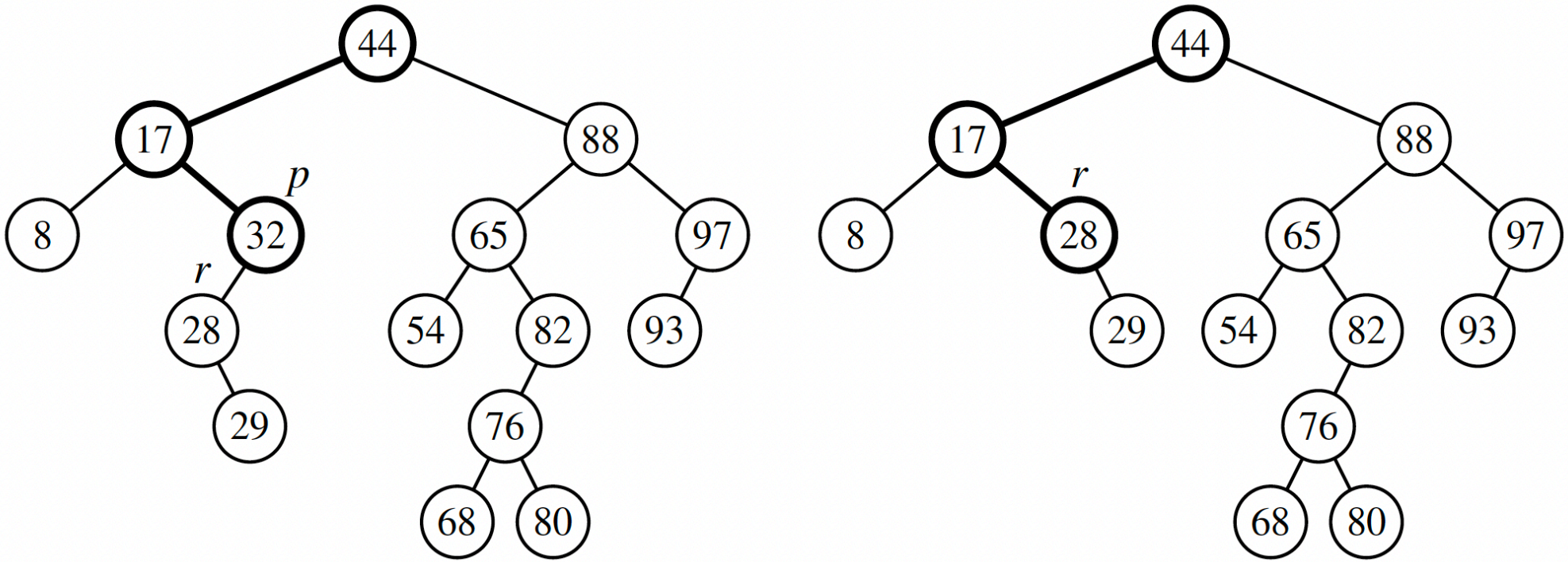


Insertion: key 68

Deletion

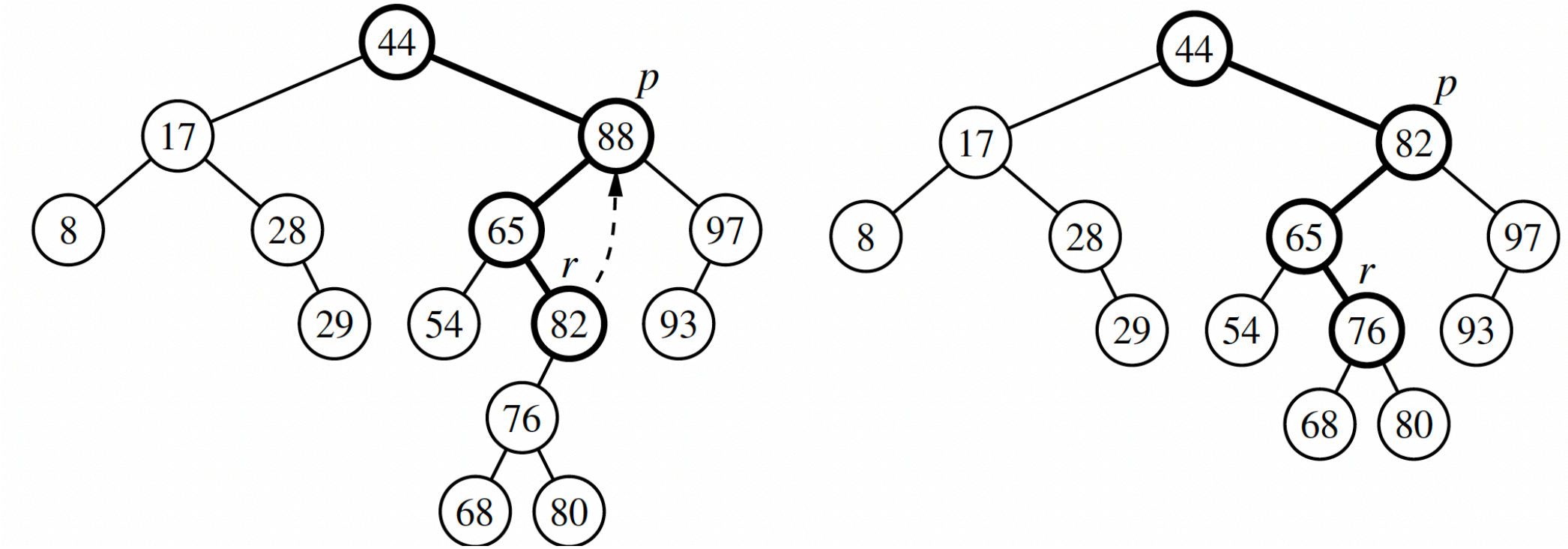
```
Algorithm TreeInsert(T, k, v):
    Input: A search key k to be associated with value v
    p = TreeSearch(T, T.root(), k)
    if k == p.key() then
        Set p's value to v
    else if k < p.key() then
        add node with item (k,v) as left child of p
    else
        add node with item (k,v) as right child of p
```

Deletion: Example



Deleting key 32

Deletion: Example



Deleting key 88

Balanced binary search tree

- Video: https://www.youtube.com/watch?v=q4fnJZr8ztY

References

1. Hetland, M. L. (2014). Python Algorithms: mastering basic algorithms in the Python Language. Apress.
2. Lee, K. D., Lee, K. D., & Steve Hubbard, S. H. (2015). *Data Structures and Algorithms with Python*. Springer.
3. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). *Data structures and algorithms in Python*. Hoboken: Wiley
4. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. MIT press.
5. Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-wesley professional.