Faculty of Information Technology, Monash University

FIT2004, S2/2016

Week 6: B-Trees and Retrieval Trees

Lecturer: Muhammad Aamir Cheema

ACKNOWLEDGMENTS

The slides are based on the material developed by Arun Konagurthu and Lloyd Allison.

Announcements

- Anonymous forum
- Assessment week 08 to be released soon
  - Questions related to B-Trees and Retrieval Trees
  - Utilize your time to implement retrieval trees in this week’s lab

Recommended readings

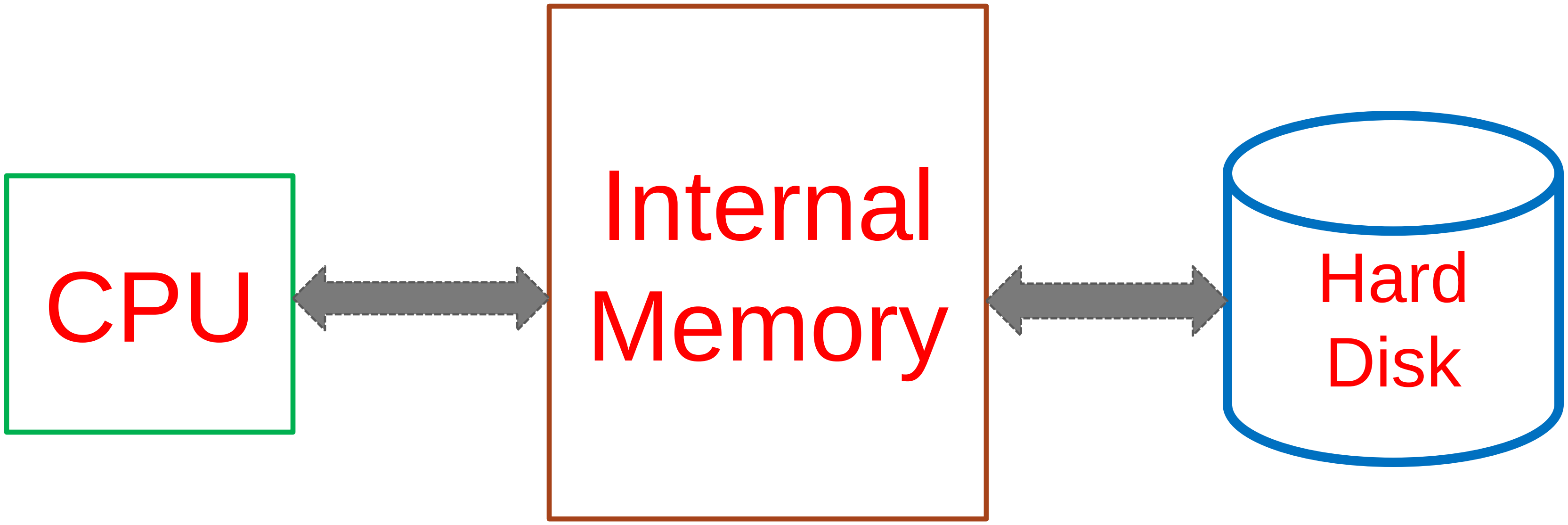
- Cormen et al. “Introduction to Algorithms" (Chapter 18)
- Tries: http://en.wikipedia.org/wiki/Trie/
- Suffix Trees: http://www.allisons.org/ll/AlgDS/Tree/Suffix/
- For a more advanced treatment of Trie and suffix trees: Dan Gusfield, Algorithms on Strings, Trees and Sequences, Cambridge University Press. (Chapter 5) - Book available in the library!

External Memory Algorithms

- So far we have assumed that the data is stored in main memory (e.g., RAM) also called internal memory, E.g.,
  - All data structures we saw so far (e.g., arrays, lists, heaps, trees, etc.) are created when the program is run and are stored in the main memory
- What if the data is too large to fit in the main memory or you want the data to be persistent (available even when the program terminates), e.g., Facebook graph contains billions of nodes, Google Maps, Monash Database etc.
  - The data is stored in secondary memory (e.g., hard disk) also called external memory
  - External memory algorithms need to read the data from the secondary storage

Internal memory vs external memory

- Internal Memory
  - Small size (e.g., 64 GB)
  - Fast access, e.g., several nanoseconds (1 ns = 10^{-9} sec)
- External memory
  - Large size (e.g., 16 TB → 16,000 GB)
  - Slow time, e.g., several milliseconds (1ms = 10^{-3} sec = 10^{6} ns)
- Bottleneck of the external memory algorithm is the number of accesses to hard disk also called I/O cost (Input/Output cost).
- Therefore, external memory algorithms are analyzed in terms of time complexity (e.g., # of operations) as well as I/O cost (e.g., # of disk accesses)



BST in External Memory

- Suppose we have a set of records to be stored on hard disk.
- One solution to allow lookups is to create a Binary Search Tree and store it in the external memory
- The total I/O cost for searching a key is $O(\log_2 N)$ disk accesses
- Can we do better?

Yes! B-Trees

B-Tree motivation

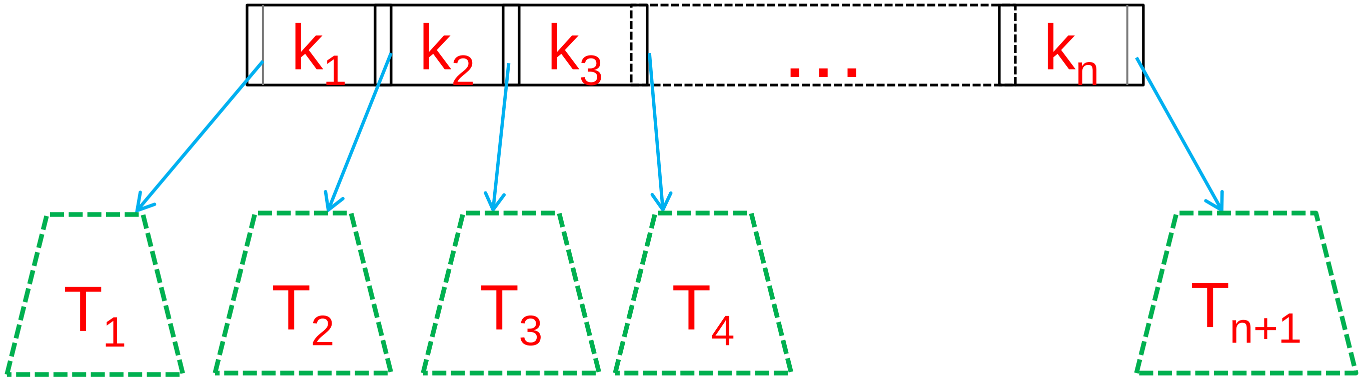
- When disk is accessed, a block of size B (e.g., 4 KB) is accessed. , i.e., instead of accessing only one value , we can access as many values as can be fit in the block of size B in a single disk access (i.e., in one I/O).
  - Key idea: BST stores only one value (and two pointers to its children) in each node. Accessing this node requires one I/O
  - What if we make the node bigger such that it can accommodate as many values and pointers as possible in a block of size B. Accessing this node will still require a single disk access but we would fetch many more values.
  - Hence, use a tree where nodes can have many children instead of at most two as in binary tree.

B-Tree Introduction

- B-trees generalize balanced search trees
- By generalization, the tree is no longer binary but has many branches per node.
- They are really powerful and are used on many mission critical systems that rely on a large amount of data stored on a secondary storage device (disks)
- Common examples are very large databases and file systems

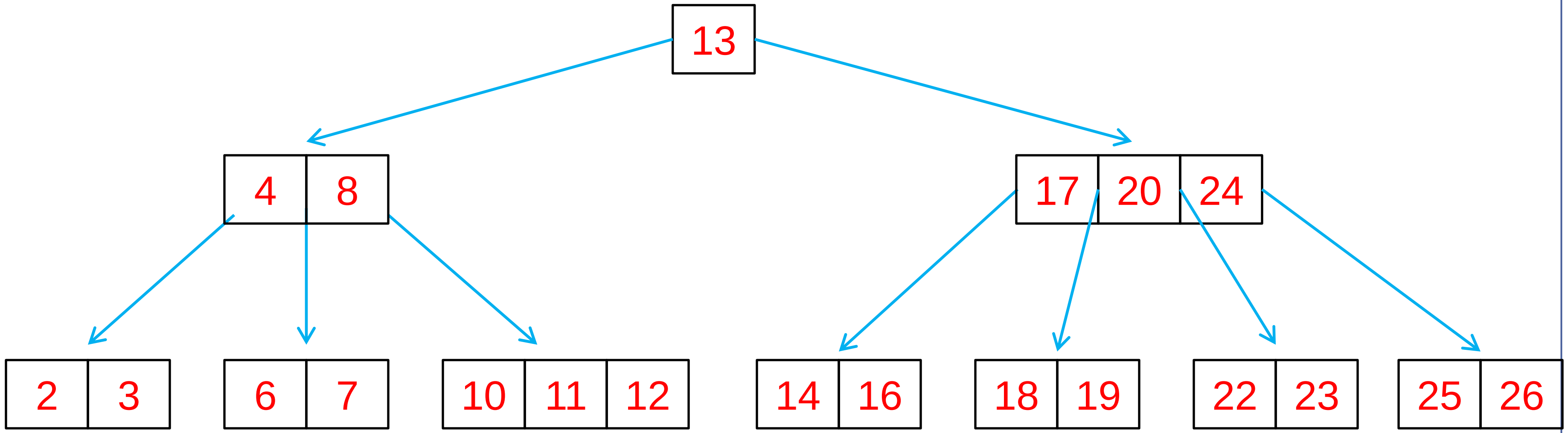
B-Tree Properties

- A B-tree is a rooted node
- An arbitrary node has n keys and has n+1 pointers to its children
- The keys are stored in the node in ascending order (i.e., $k_1 \leq k_2 \leq k_3 \leq \ldots \leq k_n$)
- All elements in a sub-tree T_i are less than equal to k_i and greater than or equal to k_{i+1}. i.e., ., $\{T_1\} \leq k_1 \leq \{T_2\} \leq k_2 \leq \ldots \leq k_n \leq \{T_{n+1}\}$
- One disk access corresponds to reading one node of the B-tree
- Each pointer in a node points to the location of the node in the hard disk



B-Tree Properties

- Each non-root node must have at least t-1 keys (t is called minimum degree)
- Each node (including root) can have at most 2t-1 keys
  - A node that has more than 2t-1 keys is called overflowed. This must be fixed
  - A non-root node that has les than t-1 keys is called underflowed. This must be fixed
  - A node is said to be full if it contains exactly 2t – 1 keys
- All leaf nodes have the same depth
- The height of a B-Tree is O($\log_t$ N) where N is the number of entries in B-Tree
  - Note that $\log_t$ N is significantly smaller than $\log_2$ N if t is not small
  - E.g., if N is 1 Billion, the height of binary search tree is at least 30, where the height of B-tree would be 3 if t is 1000 or 5 if t is 100



t = 3: Each node contains at most 5 keys

Each non-root node contains at least 2 keys

Searching in B-Tree

Searching is very similar to the searching in Binary Search Tree

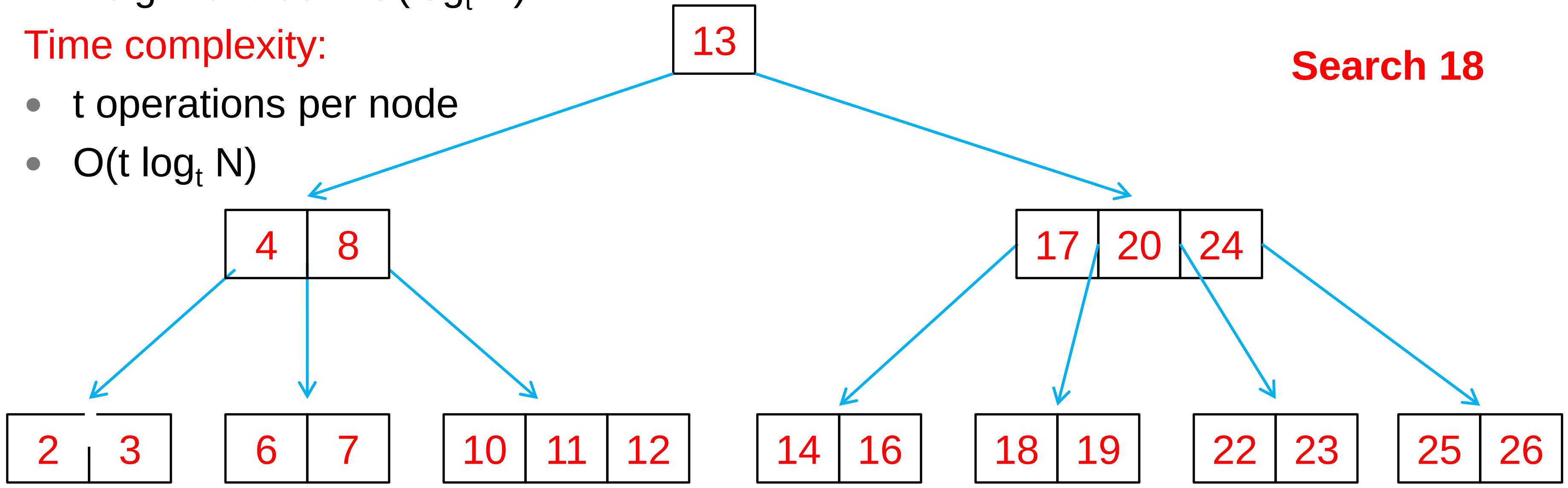
1. Load the root-node from the hard disk
2. Look at each key in the node and find the appropriate node N to continue the search
3. Load the node N from the hard disk and go to step 2 until the element is found

I/O cost (number of disk accesses):

- Height of tree = O($\log_t$ N)

Time complexity:

- t operations per node
- O(t $\log_t$ N)



t = 3: Each node contains at most 5 keys
Each non-root node contains at least 2 keys

Inserting in B-Tree

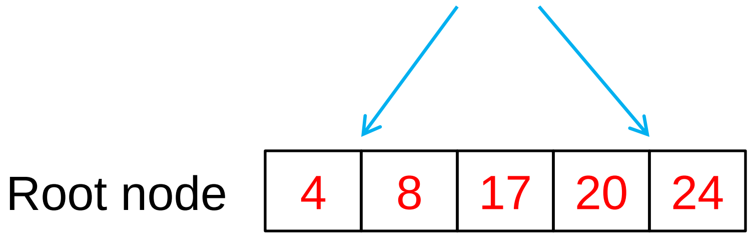
First, we explain how the full nodes are split.

Splitting a full root node:

- Choose the median entry of the node N and create a new root node R containing the median
- Split N into 2 halves based on median and add pointers

Is it possible that after splitting, one of the child nodes is underflowed?

No! A full node contains 2t -1 keys. After splitting, the root node contains 1 key and each of the split node contains exactly t-1 keys. Thus, none of the nodes is underflowed (thus satisfying the B-tree properties).



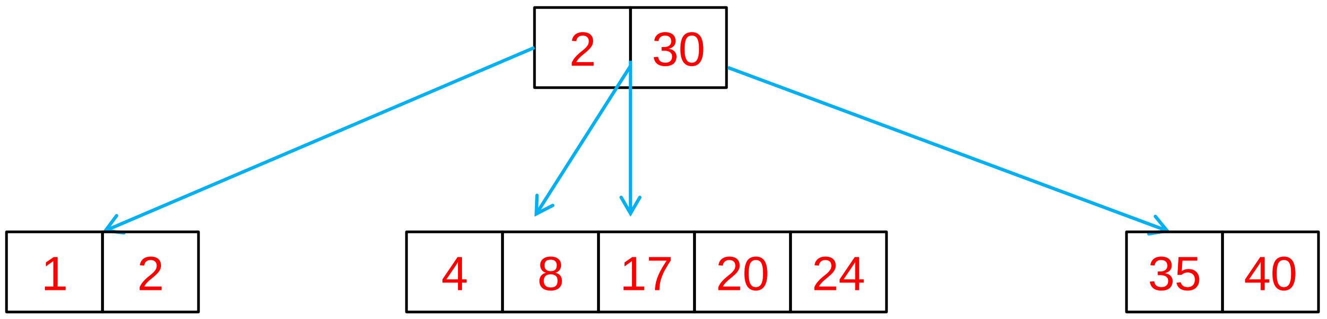
t = 3: Each node contains at most 5 keys

Each non-root node contains at least 2 keys

Inserting in B-Tree

Splitting a full non-root node

- Move the median to the parent node
- Split the node into two halves and add pointers



t = 3: Each node contains at most 5 keys
Each non-root node contains at least 2 keys

Inserting in B-Tree

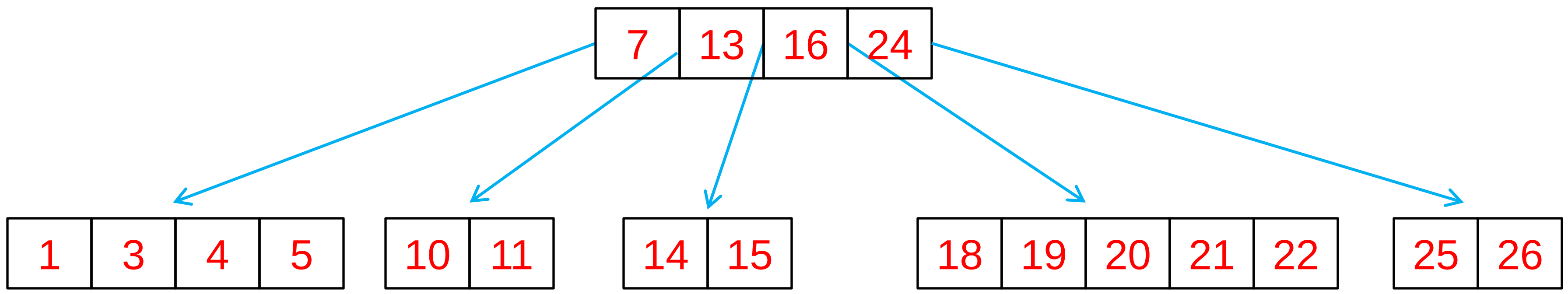
Inserting a key x

1. Traverse the tree as if we are searching for x
2. If a full node is retrieved during traversal, split it
3. When a non-full leaf node is reached, insert x in it

Example: Insert 2

- Root is not full, access its relevant child
- The leaf is not full, insert 2 in it

Insert 2



t = 3: Each node contains at most 5 keys
Each non-root node contains at least 2 keys

Inserting in B-Tree

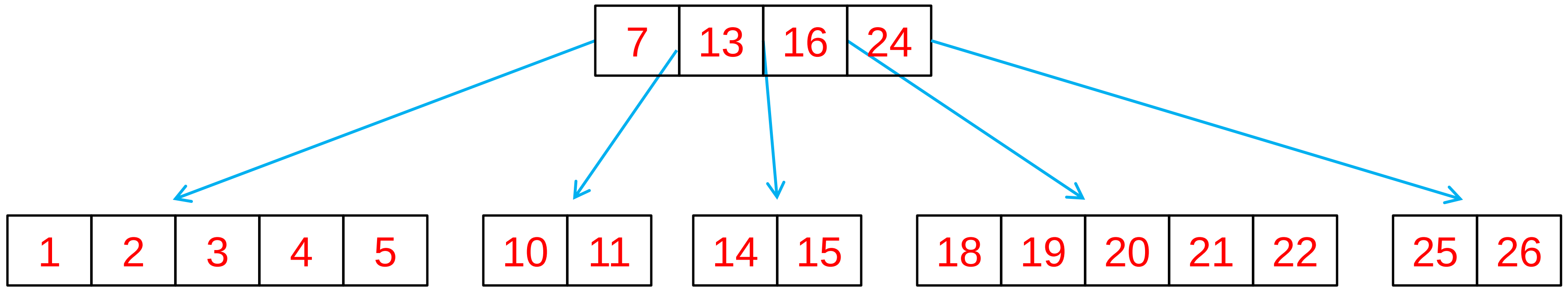
Inserting a key x

1. Traverse the tree as if we are searching for x
2. If a full node is retrieved during traversal, split it
3. When a non-full leaf node is reached, insert x in it

Example: Insert 2

- Root is not full, access its relevant child
- The leaf is not full, insert 2 in it

Insert 2



t = 3: Each node contains at most 5 keys
Each non-root node contains at least 2 keys

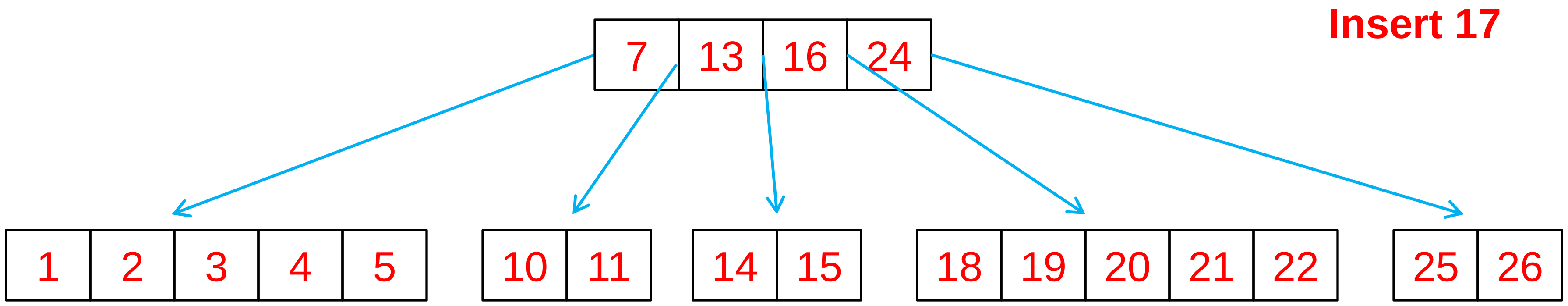
Inserting in B-Tree

Inserting a key x

1. Traverse the tree as if we are searching for x
2. If a full node is retrieved during traversal, split it
3. When a non-full leaf node is reached, insert x in it

Example: Insert 17

- Root is not full, access its relevant child
- The leaf is full, split it
    - i.e., move 20 to parent and add pointers



t = 3: Each node contains at most 5 keys
Each non-root node contains at least 2 keys

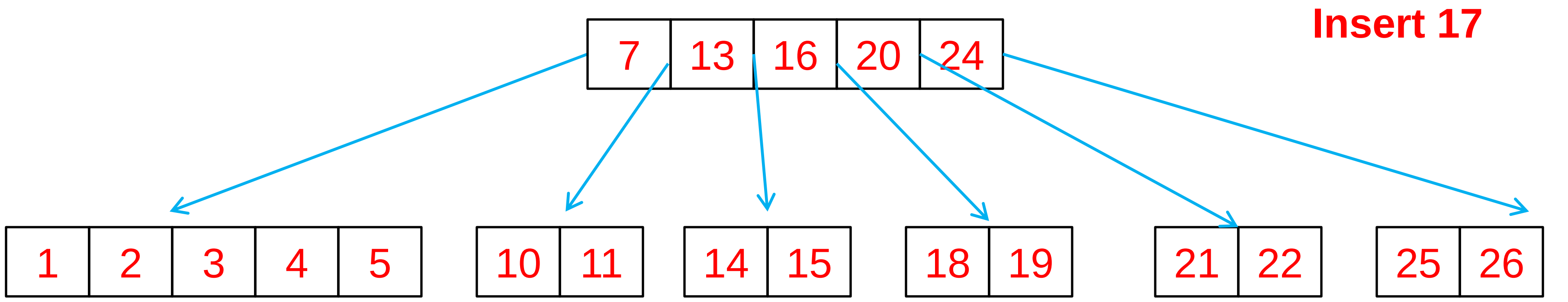
Inserting in B-Tree

Inserting a key x

1. Traverse the tree as if we are searching for x
2. If a full node is retrieved during traversal, split it
3. When a non-full leaf node is reached, insert x in it

Example: Insert 17

- Root is not full, access its relevant child
- The leaf is full, split it
  - Move 20 to parent and add pointers



t = 3: Each node contains at most 5 keys
Each non-root node contains at least 2 keys

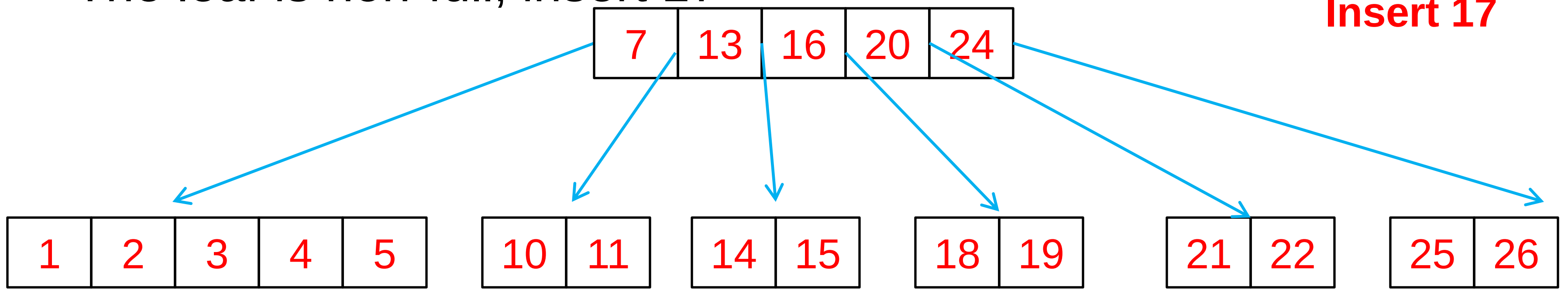
Inserting in B-Tree

Inserting a key x

1. Traverse the tree as if we are searching for x
2. If a full node is retrieved during traversal, split it
3. When a non-full leaf node is reached, insert x in it

Example: Insert 17

- Root is not full, access its relevant child
- The leaf is full, split it
  - Move 20 to parent and add pointers
- Access the relevant node
- The leaf is non-full, insert 17



t = 3: Each node contains at most 5 keys
Each non-root node contains at least 2 keys

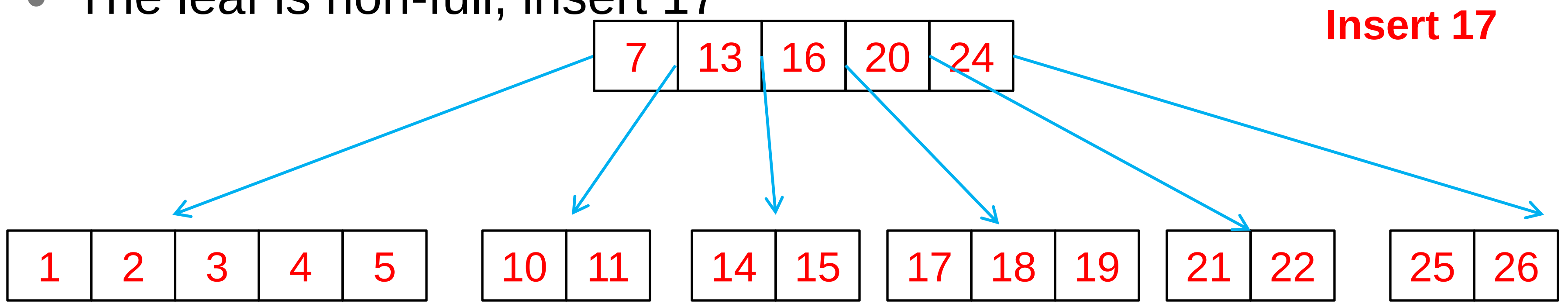
Inserting in B-Tree

Inserting a key x

1. Traverse the tree as if we are searching for x
2. If a full node is retrieved during traversal, split it
3. When a non-full leaf node is reached, insert x in it

Example: Insert 17

- Root is not full, access its relevant child
- The leaf is full, split it
  - Move 20 to parent and add pointers
- Access next relevant node
- The leaf is non-full, insert 17

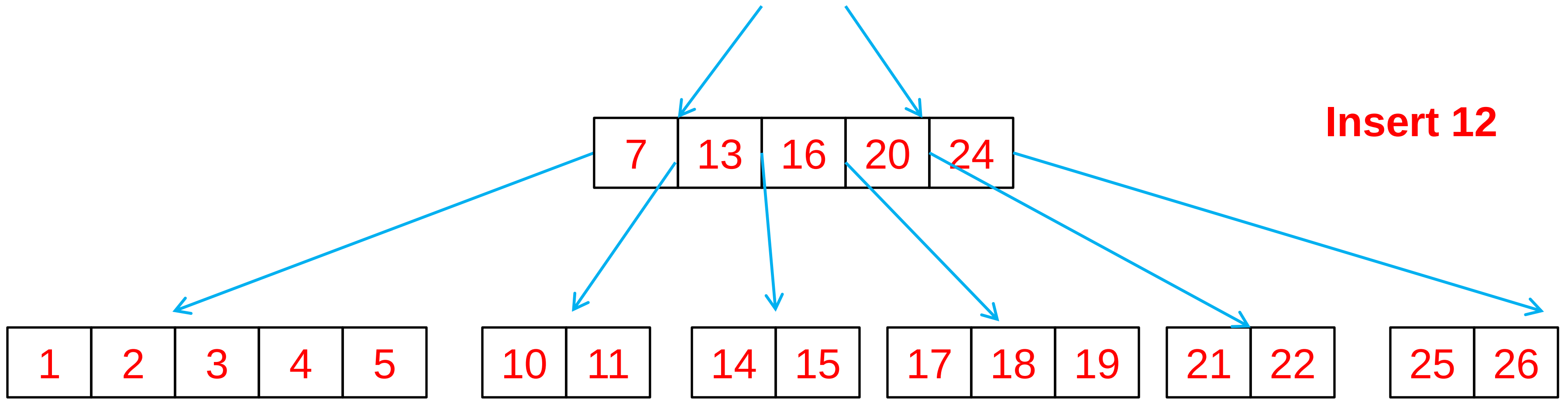


t = 3: Each node contains at most 5 keys
Each non-root node contains at least 2 keys

Inserting in B-Tree

Example: Insert 12

- Root is full, split it
  - i.e., move median to a new root node and add pointers

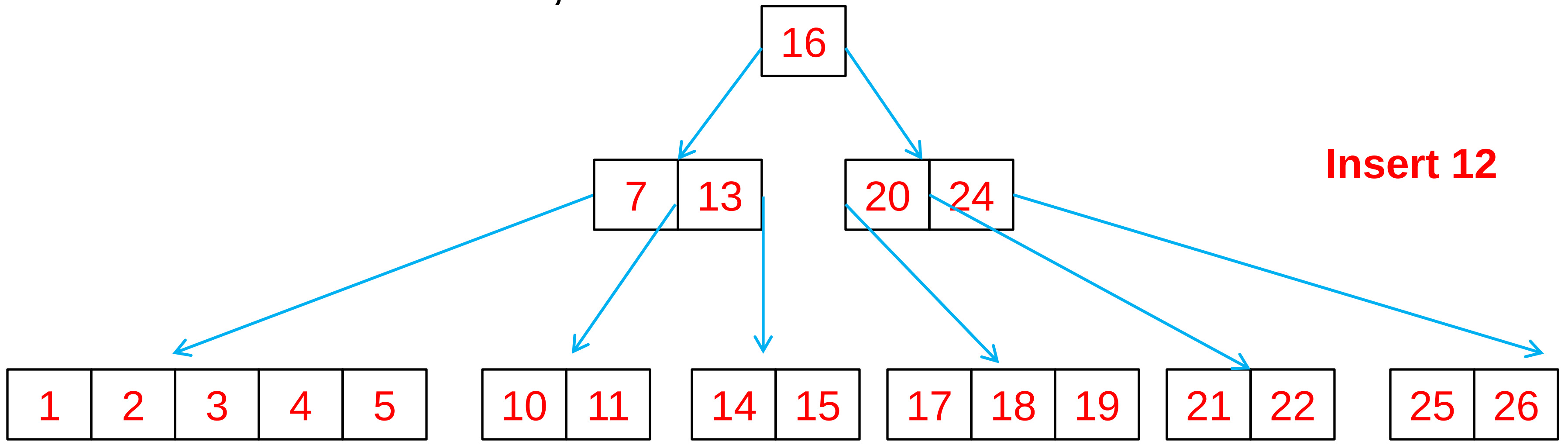


t = 3: Each node contains at most 5 keys
Each non-root node contains at least 2 keys

Inserting in B-Tree

Example: Insert 12

- Root is full, split it
  - i.e., move median to a new root node and add pointers
- Access the relevant node
- It cannot be full, access next relevant node
- It is a non-full leaf, insert here

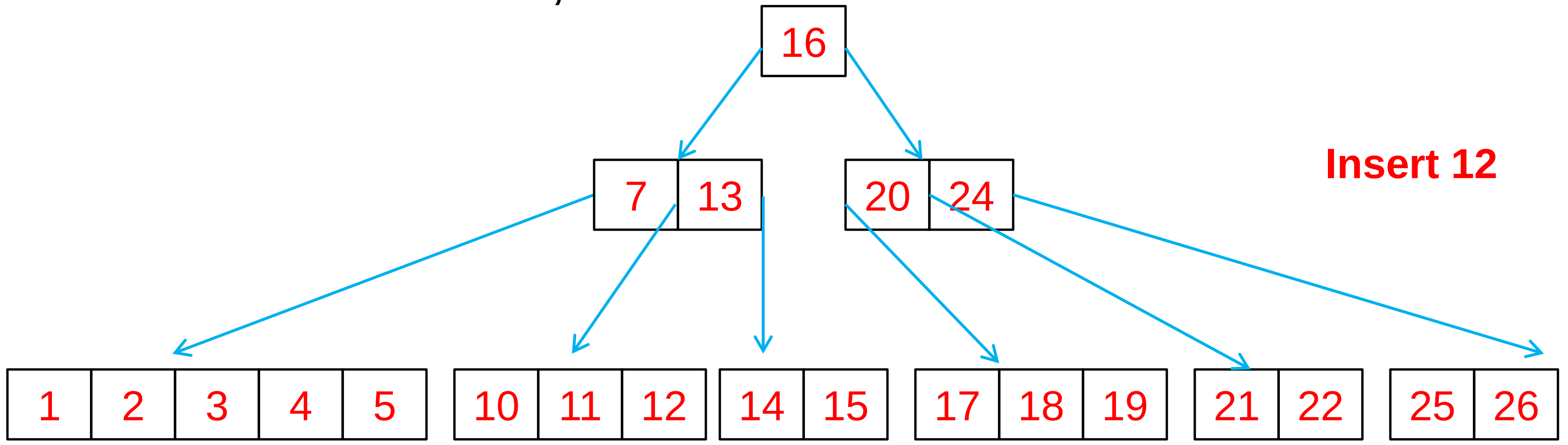


t = 3: Each node contains at most 5 keys
Each non-root node contains at least 2 keys

Inserting in B-Tree

Example: Insert 12

- Root is full, split it
  - i.e., move median to a new root node and add pointers
- Access the relevant node
- It cannot be full, access next relevant node
- It is a non-full leaf, insert here

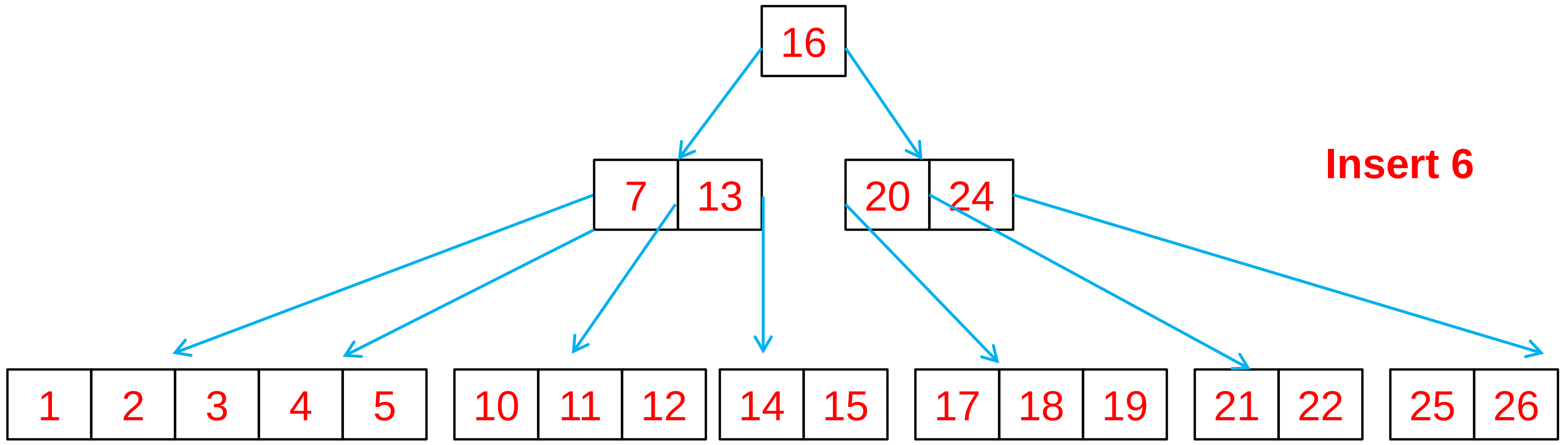


t = 3: Each node contains at most 5 keys
Each non-root node contains at least 2 keys

Inserting in B-Tree

Example: Insert 6

- Root is non-full, access next relevant node
- It is not full, access next relevant node
- It is full, split it
  - i.e., move median to parent node and add pointers

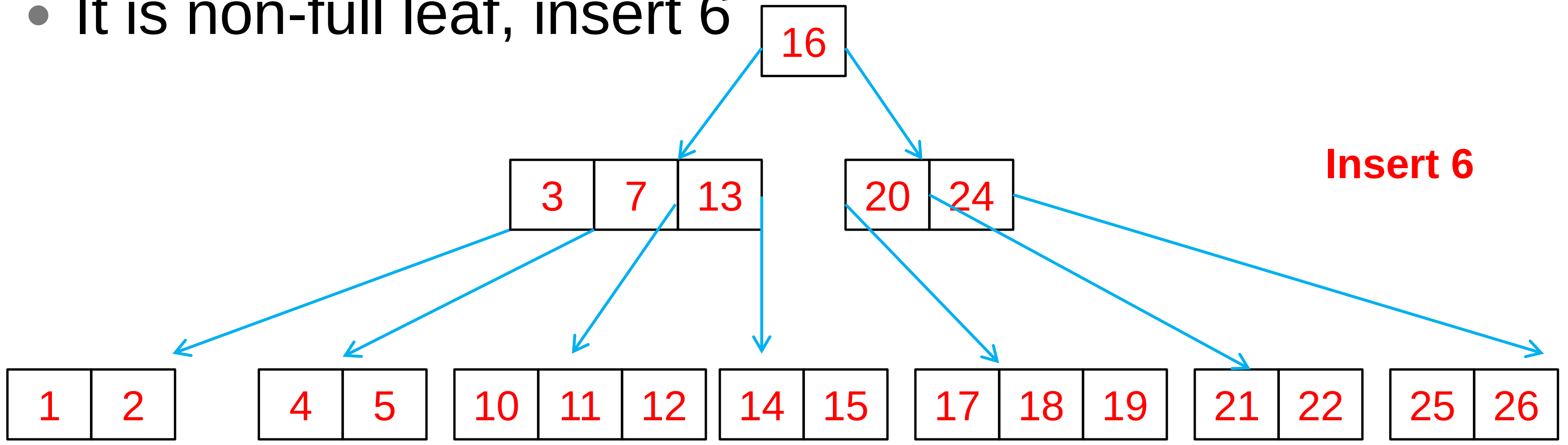


t = 3: Each node contains at most 5 keys
Each non-root node contains at least 2 keys

Inserting in B-Tree

Example: Insert 6

- Root is non-full, access next relevant node
- It is not full, access next relevant node
- It is full, split it
  - i.e., move median to parent node and add pointers
- Access relevant node
- It is non-full leaf, insert 6

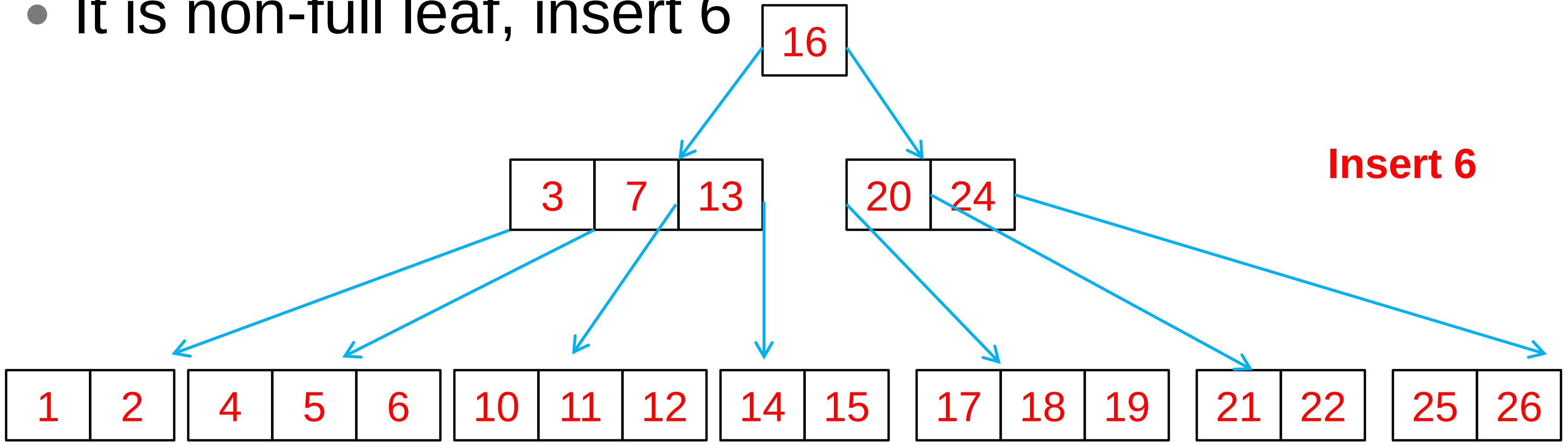


t = 3: Each node contains at most 5 keys
Each non-root node contains at least 2 keys

Inserting in B-Tree

Example: Insert 6

- Root is non-full, access next relevant node
- It is not full, access next relevant node
- It is full, split it
  - i.e., move median to parent node and add pointers
- Access relevant node
- It is non-full leaf, insert 6



t = 3: Each node contains at most 5 keys
Each non-root node contains at least 2 keys

Complexity of Insertion

Inserting a key x

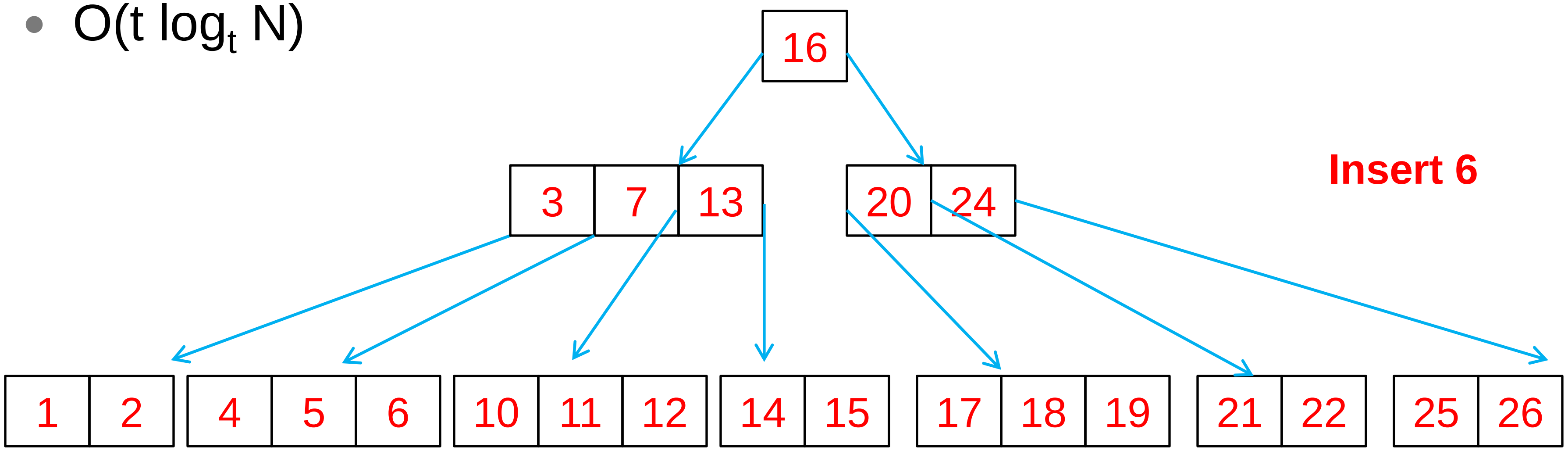
1. Traverse the tree as if we are searching for x
2. If a full node is retrieved during traversal, split it
3. When a non-full leaf node is reached, insert x in it

I/O Cost (number of disk accesses):

- Height of the tree= $O(\log_t N)$

Time complexity:

- $O(t \log_t N)$



t = 3: Each node contains at most 5 keys
Each non-root node contains at least 2 keys

Deletion in B-Tree

Terminology:

Recall that each non-root node must have at least t-1 elements

- A node is called “underflowed” if it has less than t-1 elements
- A node is called “rich” if it has at least t elements (i.e., it can give away one element without getting underflowed”)
- A node is called “poor” it has exactly t-1 elements (i.e., deletion will cause this node to be overflowed)

Delete a key x

Case 1: x belongs to a rich leaf node

Case 2: x belongs to a rich non-leaf node

Case 3: the top-down traversal to x passes through a poor non-root node

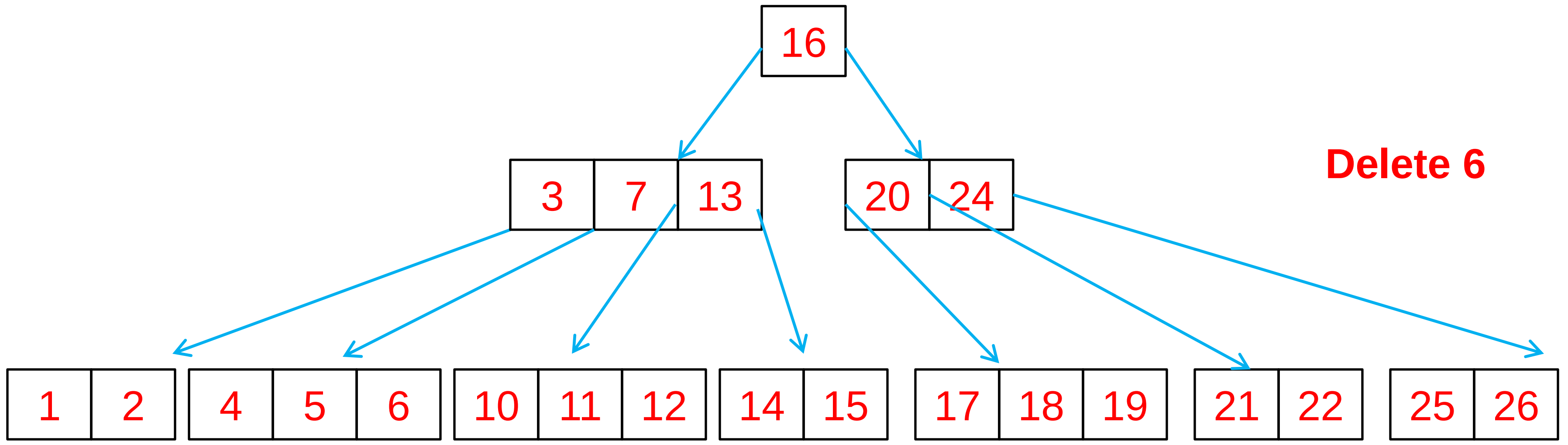
Deletion: Case 1

Case 1: x belongs to a rich leaf node

This is a trivial case

1. Delete x from the leaf node

Example: Delete 6



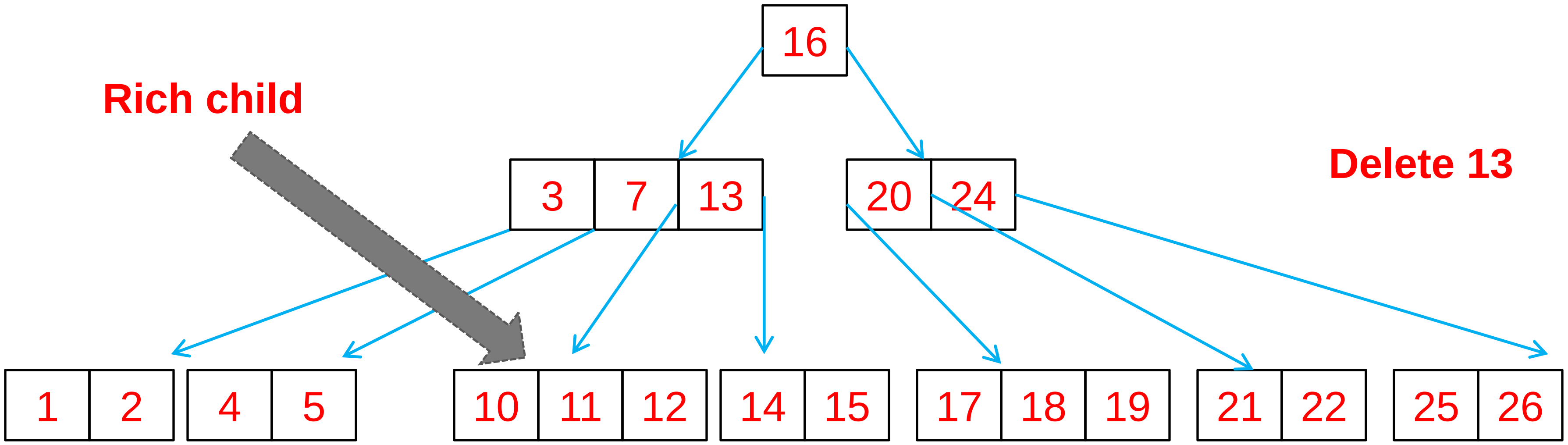
t = 3 →Rich node: > 2 keys, Poor node: 2 keys, overflowed: < 2
Each non-root node must contain at least 2 keys

Deletion: Case 2

Case 2: x belongs to a rich non leaf node

- Case 2a: One of the children of x is rich
- Case 2b: Both children of x are poor

Example: delete 13



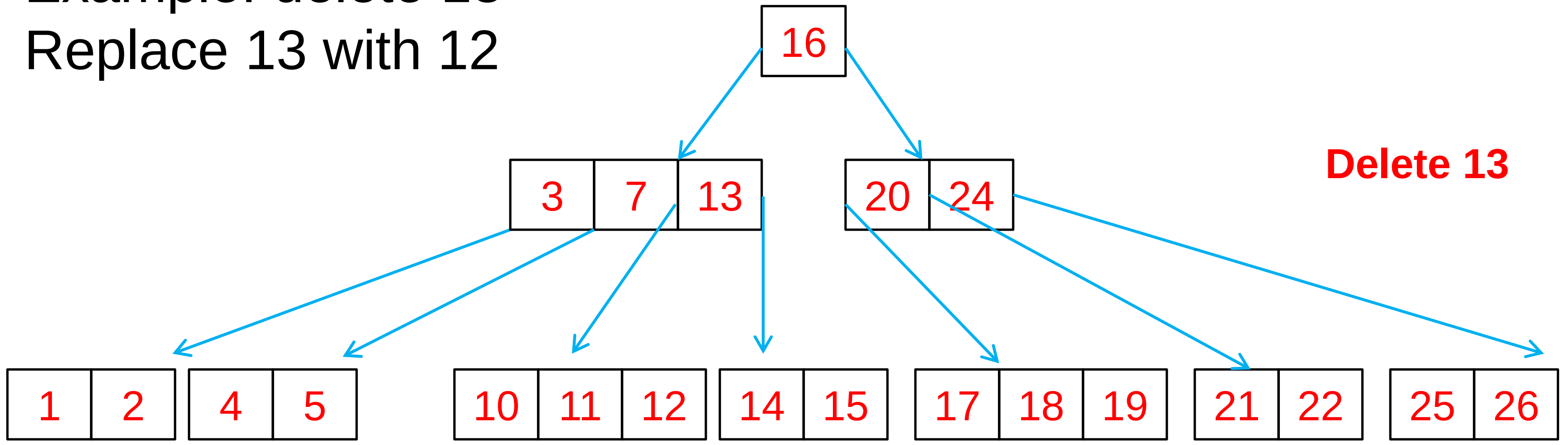
t = 3 →Rich node: > 2 keys, Poor node: 2 keys, overflowed: < 2
Each non-root node must contain at least 2 keys

Deletion: Case 2

Case 2a: One of the children is rich
(handle this case by borrowing a value from a rich child)

- If left child L is rich
  - Find the largest element t in the subtree rooted at L
  - Delete t and replace x with t
- Else, if the right child R is rich
  - Find the smallest element t in the subtree rooted at R
  - Delete t and replace x with t

Example: delete 13
Replace 13 with 12



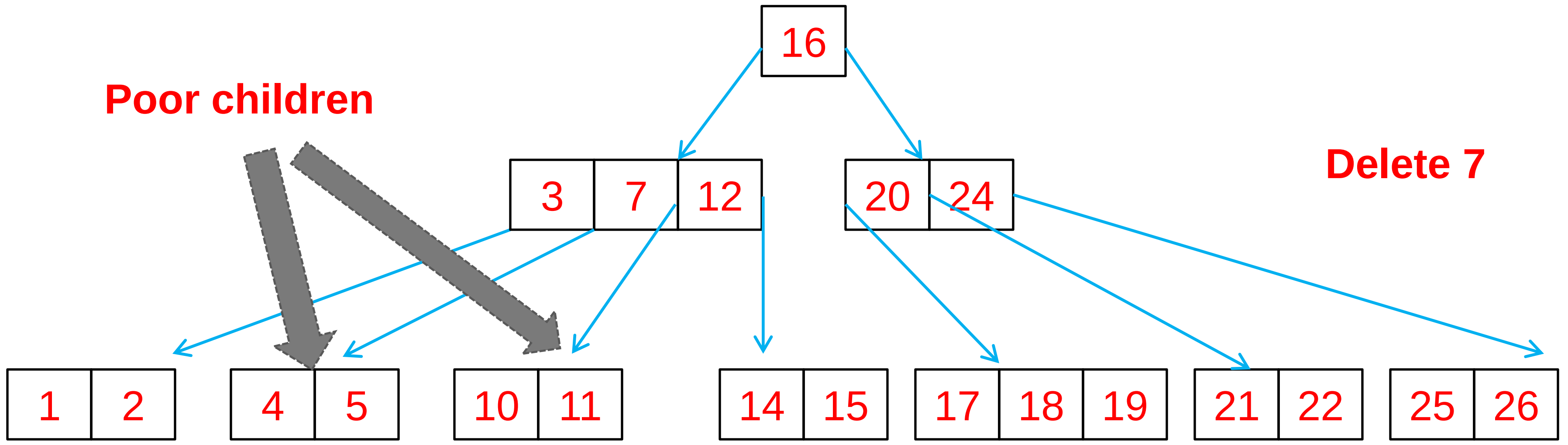
t = 3 →Rich node: > 2 keys, Poor node: 2 keys, overflowed: < 2
Each non-root node must contain at least 2 keys

Deletion: Case 2

Case 2: x belongs to a rich non leaf node

- Case 2a: One of the children is rich
- Case 2b: Both children are poor

Example: delete 7



t = 3 →Rich node: > 2 keys, Poor node: 2 keys, overflowed: < 2
Each non-root node must contain at least 2 keys

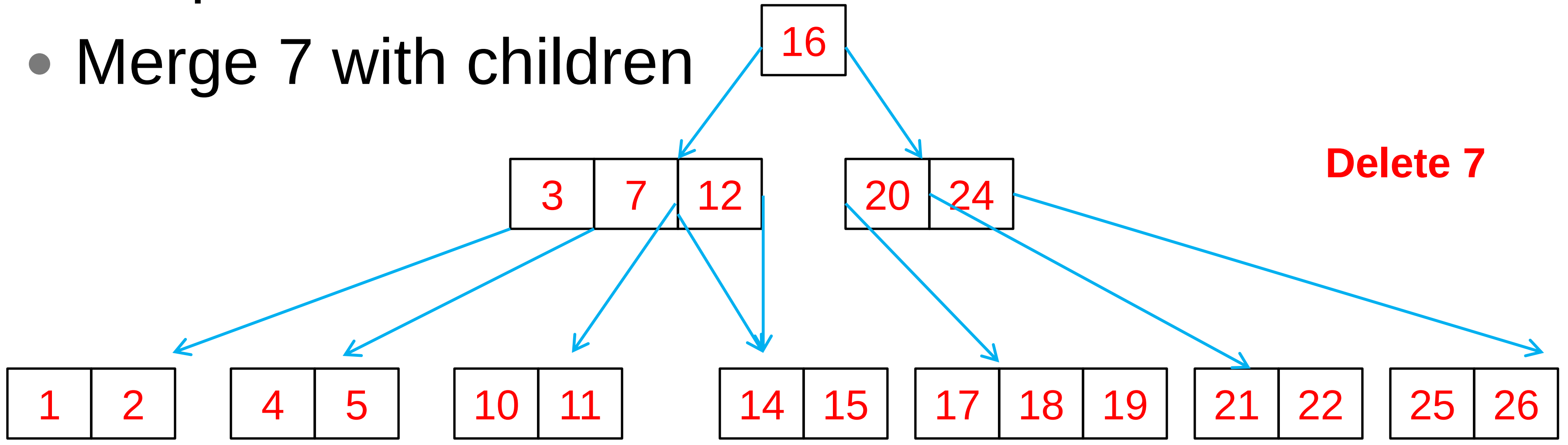
Deletion: Case 2

Case 2b: Both children are poor

- Merge parent element x with the child nodes (this cannot cause overflow)
- Then, recursively delete x from the new node

Example: delete 7

- Merge 7 with children



t = 3 →Rich node: > 2 keys, Poor node: 2 keys, overflowed: < 2
Each non-root node must contain at least 2 keys

Deletion: Case 2

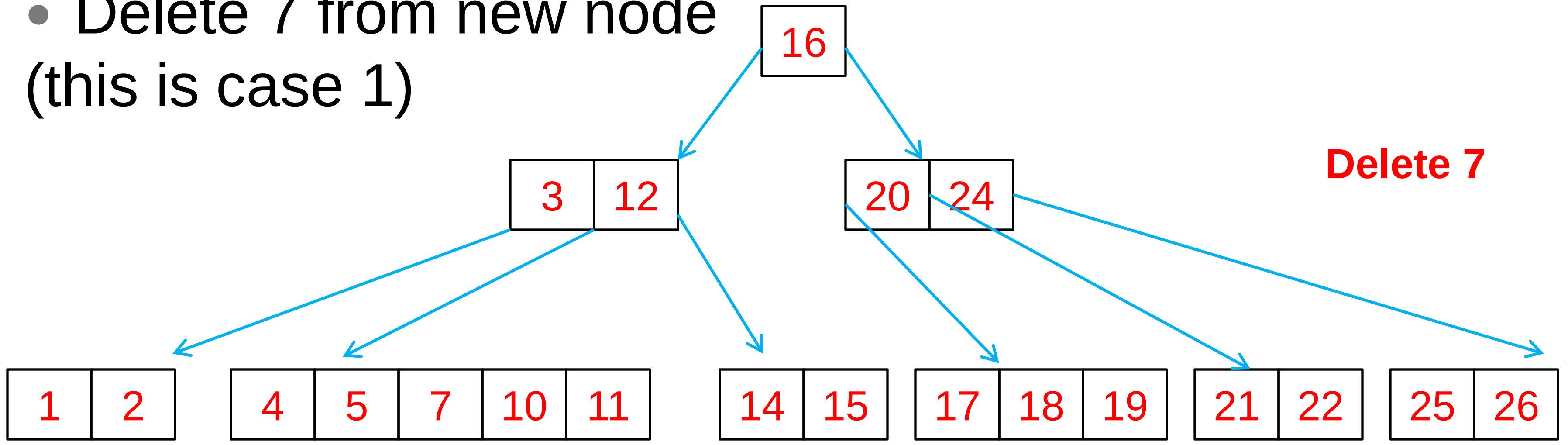
Case 2b: Both children are poor

- Merge parent element x with the child nodes (this cannot cause overflow)
- Then, recursively delete x from the new node

Example: delete 7

- Merge 7 with children
- Delete 7 from new node

(this is case 1)



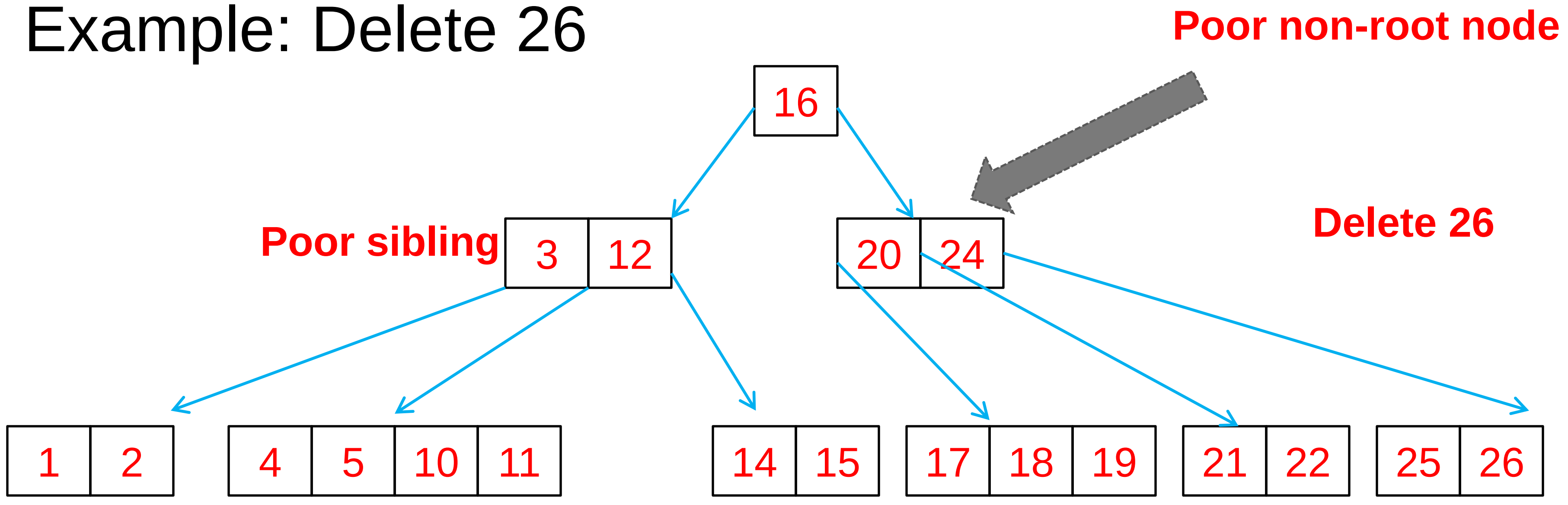
t = 3 →Rich node: > 2 keys, Poor node: 2 keys, overflowed: < 2
Each non-root node must contain at least 2 keys

Deletion: Case 3

Case 3: Top-down search to x passes through a poor non-root node N

- Case 3a: At least one immediate sibling of N is rich
- Case 3b: The immediate sibling(s) is/are poor

Example: Delete 26



t = 3 →Rich node: > 2 keys, Poor node: 2 keys, overflowed: < 2
Each non-root node must contain at least 2 keys

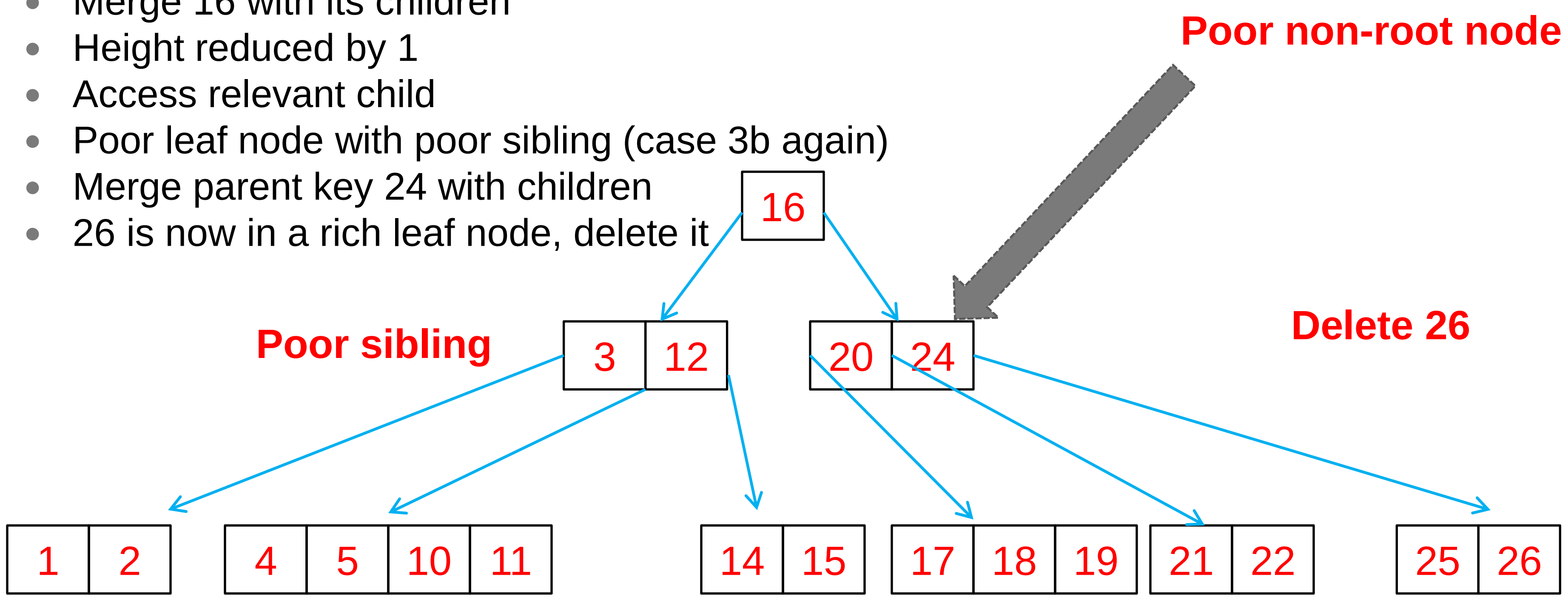
Deletion: Case 3

Case 3b: A poor non-root node N with poor siblings

- Parent key merged with children (Family reunion)
- Continue to next relevant nodes until x is reached and is in a rich node (case 1 or case 2) then delete x

Example: Delete 26

- Merge 16 with its children
- Height reduced by 1
- Access relevant child
- Poor leaf node with poor sibling (case 3b again)
- Merge parent key 24 with children
- 26 is now in a rich leaf node, delete it



t = 3 →Rich node: > 2 keys, Poor node: 2 keys, overflowed: < 2

Each non-root node must contain at least 2 keys

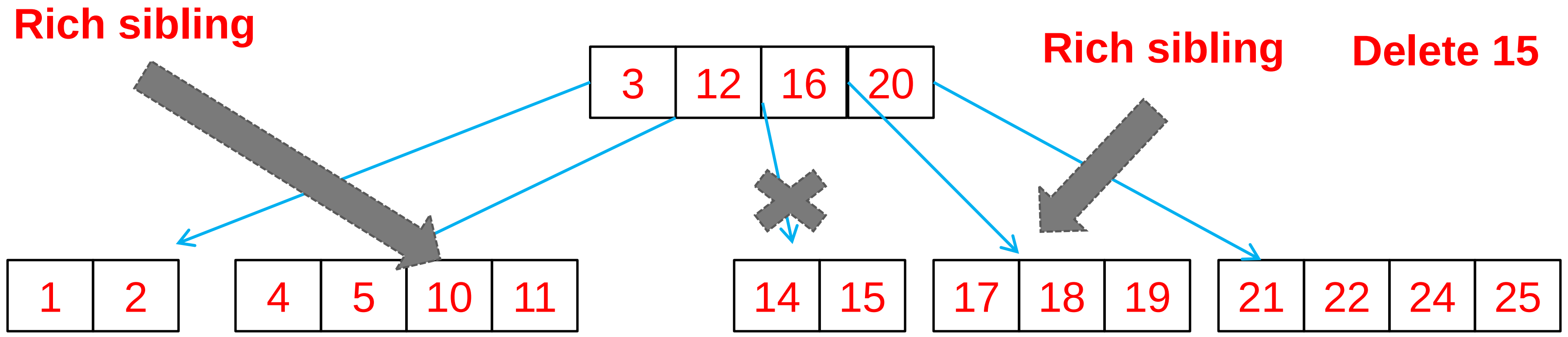
Deletion: Case 3

Case 3: Top-down search to x passes through a poor non-root node N

- Case 3a: At least one immediate sibling of N is rich
- Case 3b: The immediate sibling(s) is/are poor

Example: Delete 15

- Access relevant child
- Poor node but has at least one rich sibling
- Borrow an element from a rich sibling (no matter left or right sibling)
- Must borrow "through" parent, i.e., parent gets one from the rich sibling and gives one to the node. (e.g., moving 11 from left child to the node is WRONG, moving 17 from right child is also WRONG)



t = 3 →Rich node: > 2 keys, Poor node: 2 keys, overflowed: < 2

Each non-root node must contain at least 2 keys

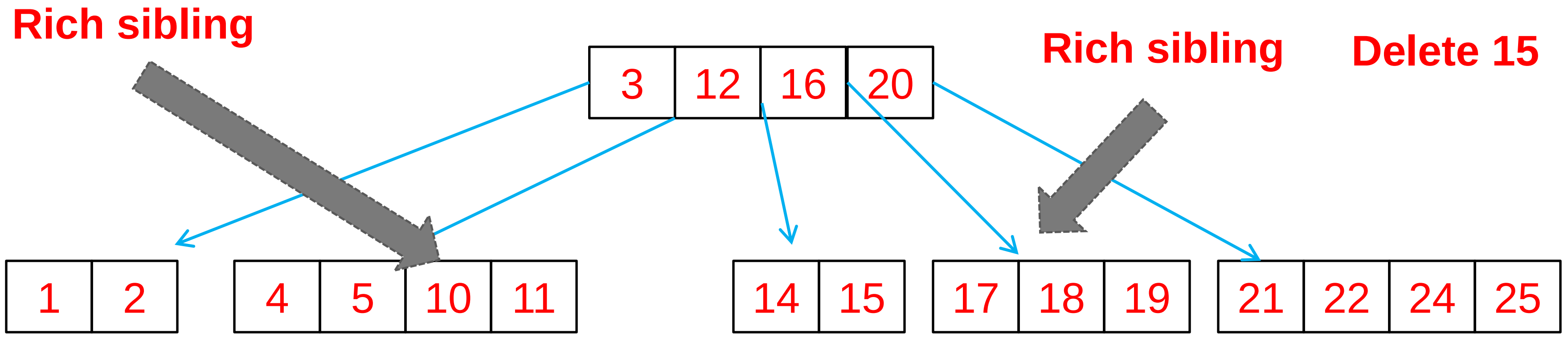
Deletion: Case 3

Case 3: Top-down search to x passes through a poor non-root node N

- Case 3a: At least one immediate sibling of N is rich
- Case 3b: The immediate sibling(s) is/are poor

Example: Delete 14

- Access relevant child
- Poor node but has at least one rich sibling
- Borrow an element from a rich sibling (no matter left or right sibling)
- Must borrow "through" parent, i.e., parent gets one from the rich sibling and gives one to the node. (e.g., moving 11 from left child to the node is WRONG, moving 17 from right child is also WRONG)
- Delete 15 from the node



t = 3 →Rich node: > 2 keys, Poor node: 2 keys, overflowed: < 2

Each non-root node must contain at least 2 keys

Complexity of Deleting from B-Tree

I/O Cost (number of disk accesses)

- $O(\log_t N)$

Time Complexity:

- $O(t \log_t N)$

Re<u>trie</u>val Trees: Introduction

Suppose you have a large text containing N strings. You want to pre-process it such that searching on this text is efficient.

Sorting based approach:

- Pre-processing: Sort the strings
- Searching: Binary search to find

Let M be the average length of strings (M can be quite large, e.g., for DNA sequences). Comparison between two strings (e.g., a < b, a==b) takes O(M).

Time complexity:

Pre-processing → O(MN log N)

Searching → O(M log N)

Can we do better?

Yes! ReTrieval Trees. E.g., A Trie allows searching in O(M) with O(MN) pre-processing cost

Trie

- ReTRIEval tree = Trie
- Often pronounced as ‘Try'.
- Trie is an N-way (or multi-way) tree, where N is the size of the alphabet
  - E.g., N=2 for binary
  - N = 26 for English letters
  - N = 4 for DNA
- In a standard Trie, all words with the shared prefix fall within the same subtree/subtrie
- In fact, it is the shortest possible tree that can be constructed such that all prefixes fall within the same subtree.

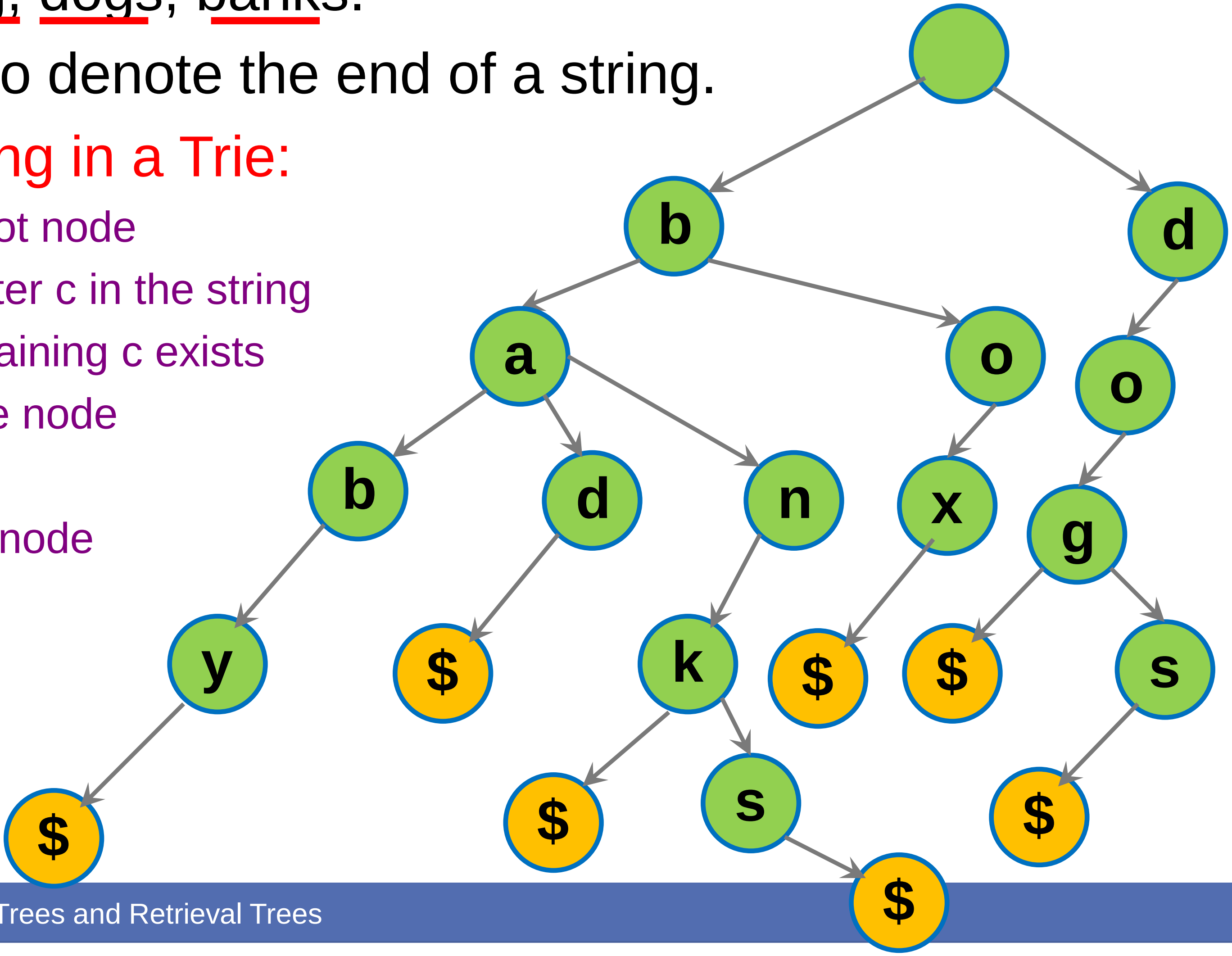
Trie Example: Insertion

Let's look at an example - a Trie that stores baby, bad, bank, box, dog, dogs, banks.

We will use $ to denote the end of a string.

Inserting a string in a Trie:

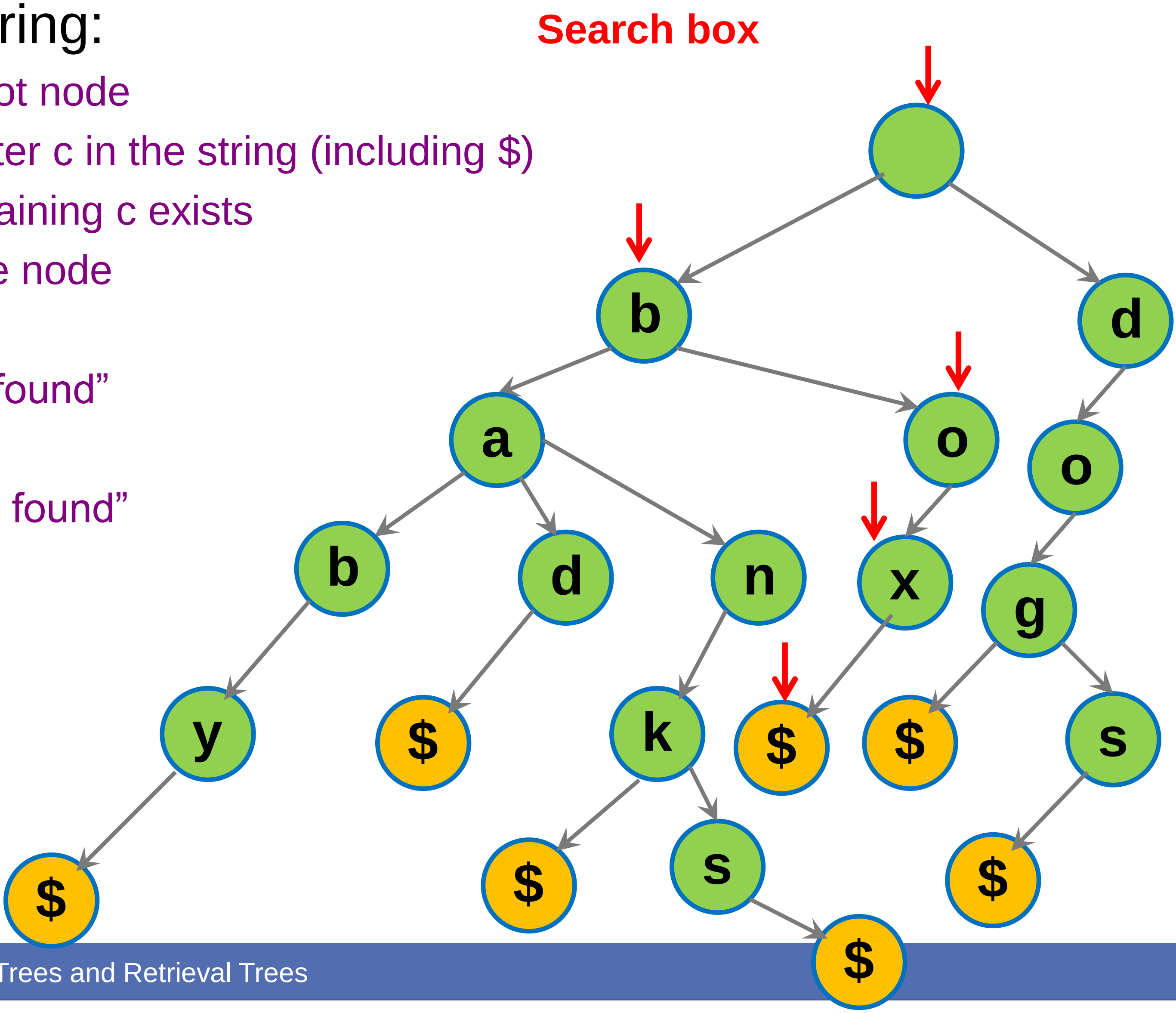
- Start from the root node
- For each character c in the string
  - If a node containing c exists
    - Move to the node
  - Else
    - Create the node
    - Move to it



Trie Example: Search

Searching a string:

- Start from the root node
- For each character c in the string (including $)
  - If a node containing c exists
    - Move to the node
    - If c == $
      - Return “found”
  - Else
    - Return “not found”

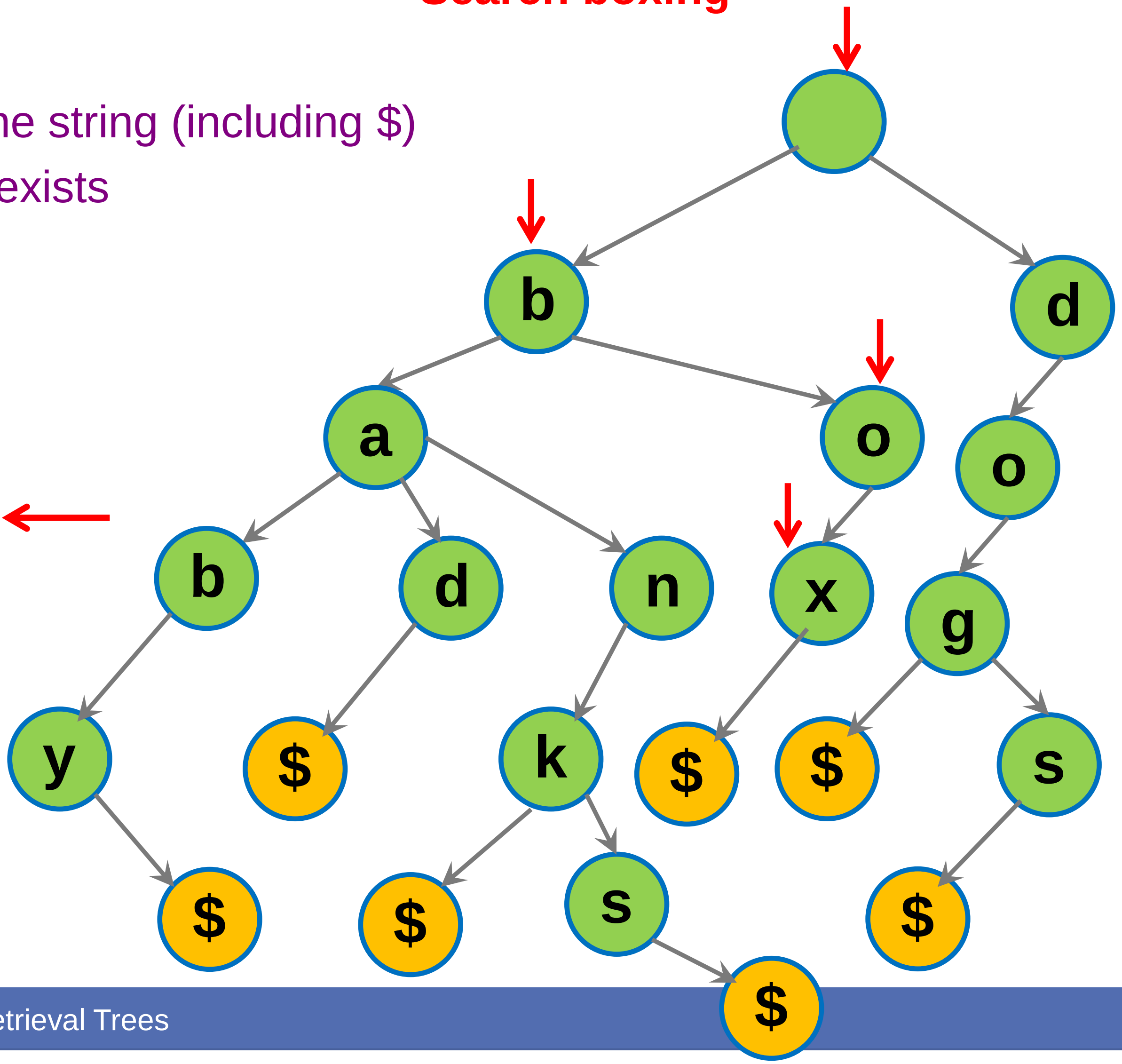


Trie Example: Search

Searching a string:

- Start from the root node
- For each character c in the string (including $)
  - If a node containing c exists
    - Move to the node
    - If c == $
      - Return "found"
  - Else
    - Return "not found" ←

Search boxing



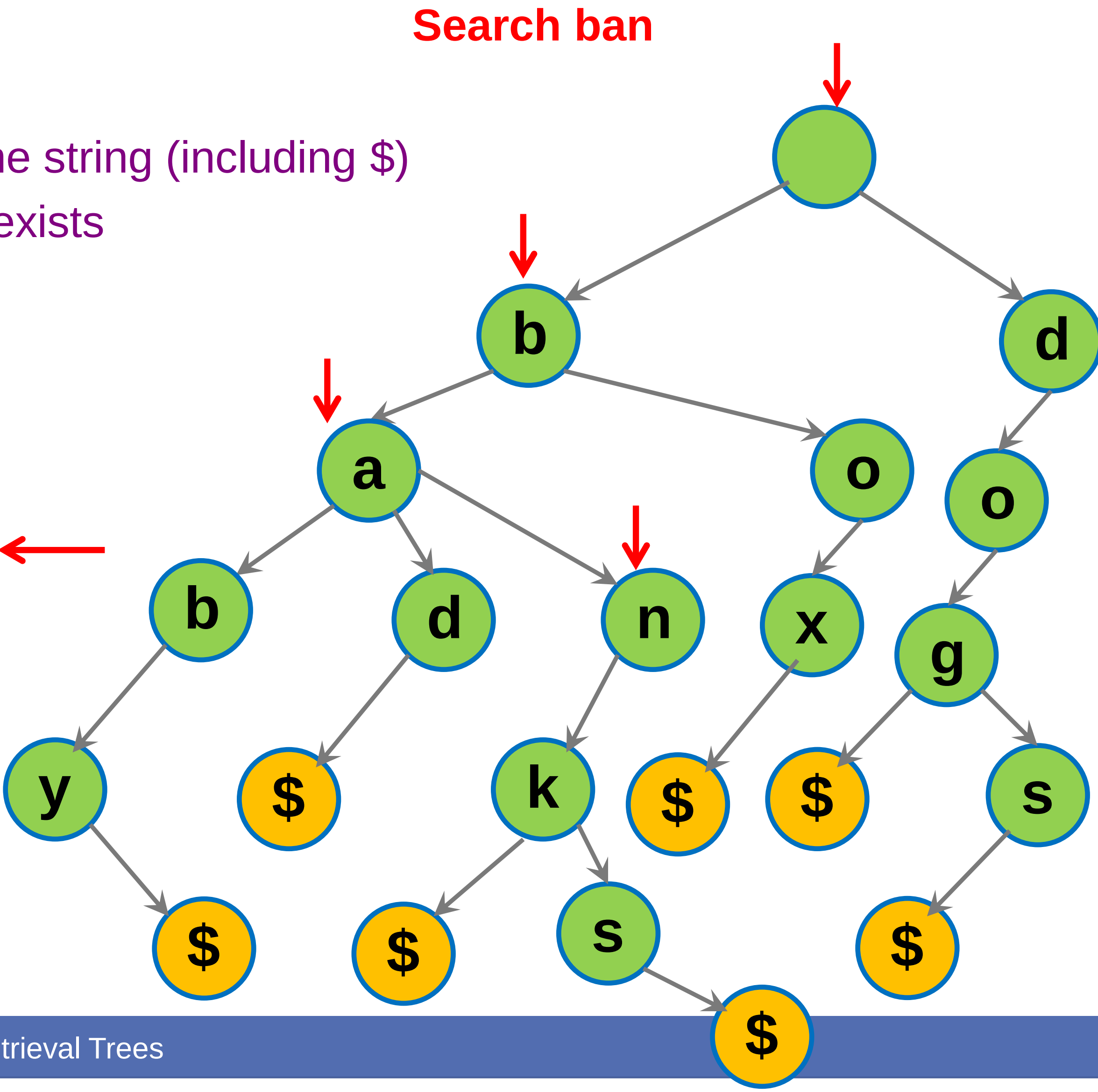
Trie Example: Search

Searching a string:

- Start from the root node
- For each character c in the string (including $)
  - If a node containing c exists
    - Move to the node
    - If c == $
      - Return “found”
  - Else
    - Return “not found” ←

Time Complexity:

- For loop runs O(M) times.
- Time to check if a node containing c exists?
  - O(1) if using an array implementation (e.g., direct-addresing)



Trie Example: Prefix Matching

Prefix matching returns every string in text that has the given string as its **prefix**.

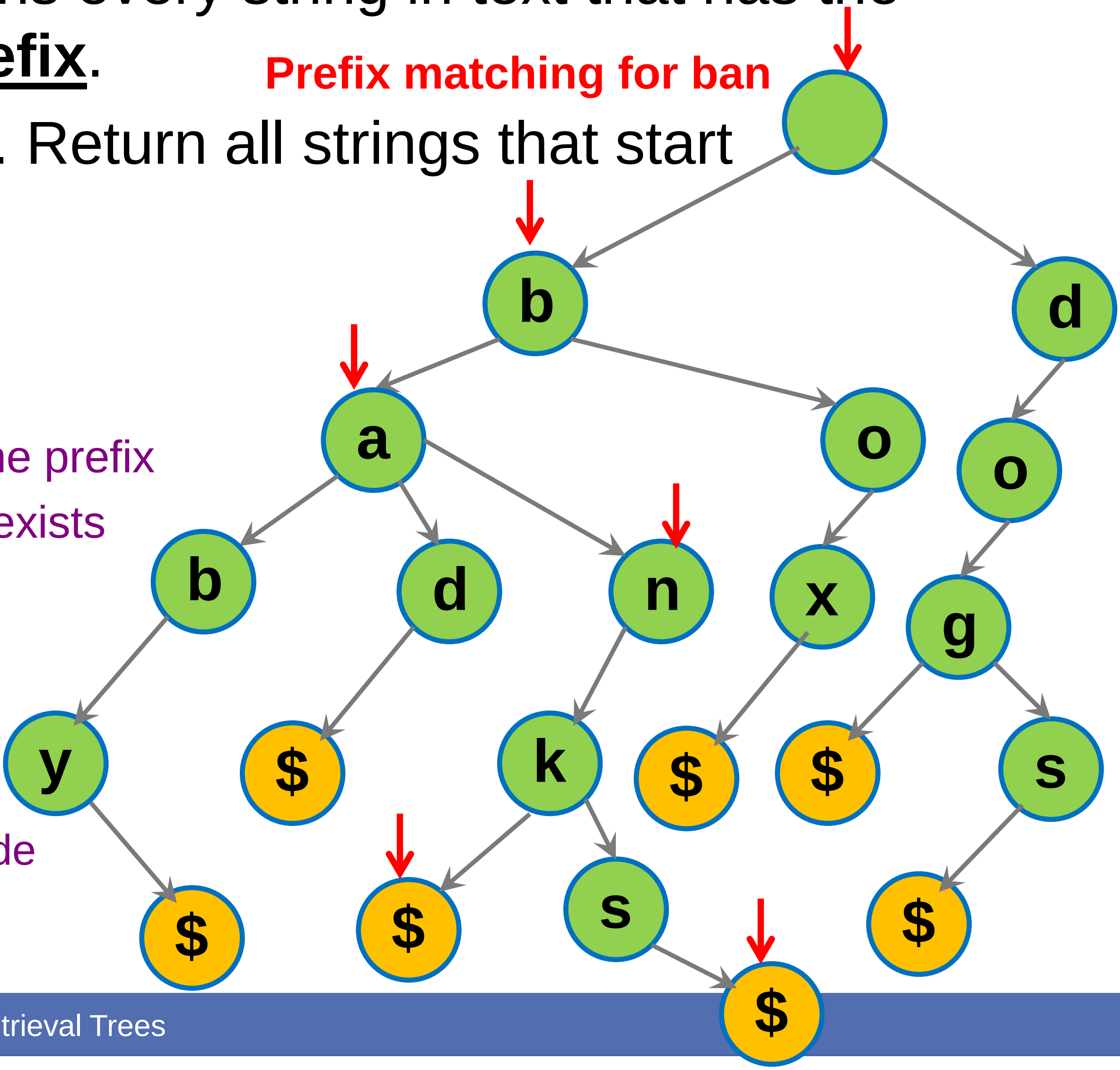
E.g., Autocompletion. Return all strings that start with “ban”

Prefix matching:

- Start from the root node
- For each character c in the prefix
  - If a node containing c exists
    - Move to the node
  - Else
    - Return “not found”
- Return all strings in the subtree rooted at the last node

Prefix for a string s[1…M] is a string s[1…X] where X≤M. (e.g., ban is a prefix of banks.)

Suffix for a string s[1…M] is a string s[X…M]. E.g., nks is a suffix of banks



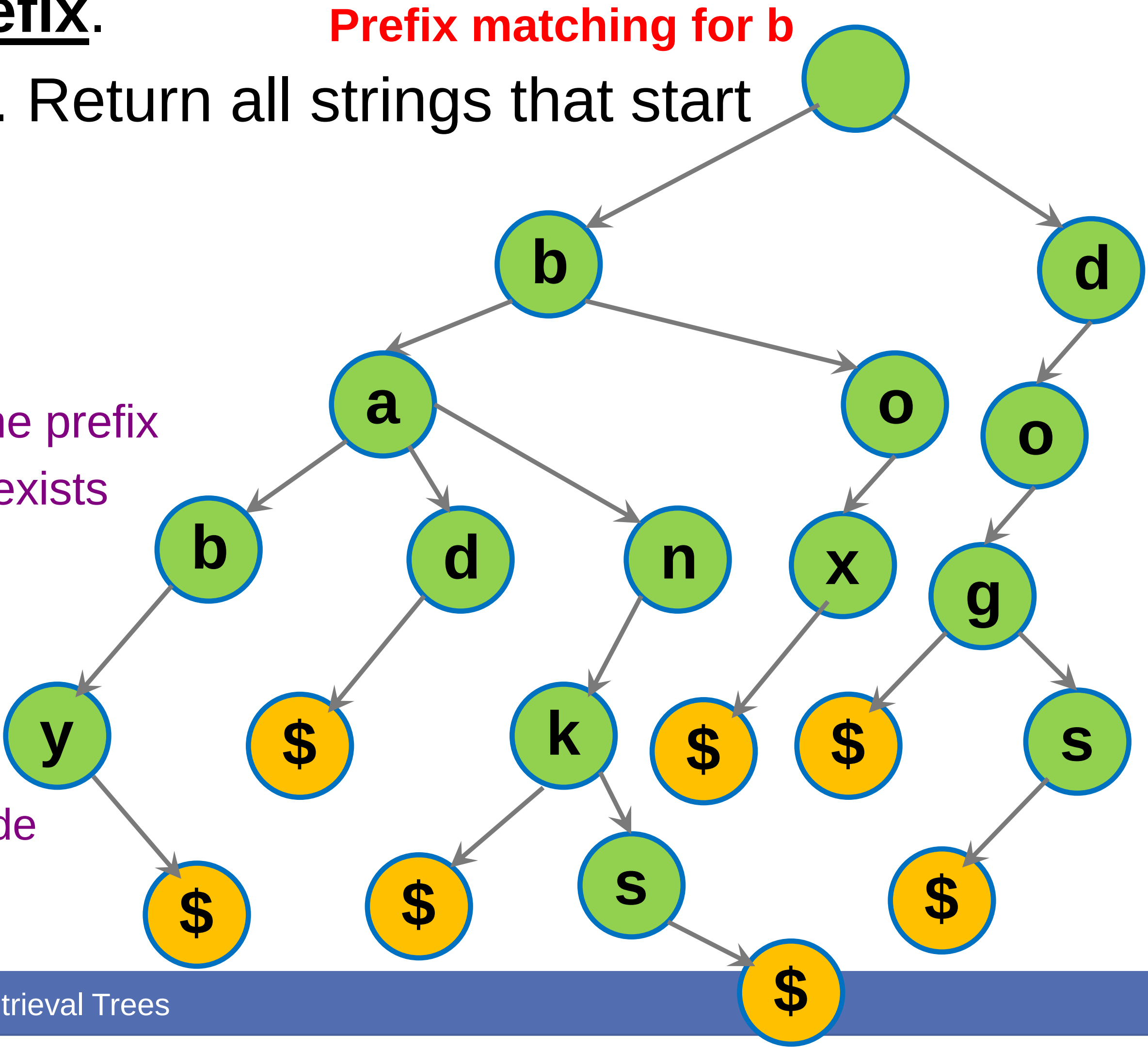
Trie Example: Prefix Matching

Prefix matching returns every string in text that has the given string as its **prefix**.

E.g., Autocompletion. Return all strings that start with "b"

Prefix matching:

- Start from the root node
- For each character c in the prefix
  - If a node containing c exists
    - Move to the node
  - Else
    - Return "not found"
- Return all strings in the subtree rooted at the last node



Implementing a Trie

Implementation using an array:

- At each node, create an array of alphabets size (e.g., 26 for English letters, 4 for DNA strings)
- If i-th node exists, add pointer to it at array[i]
- Otherwise, array[i] = Nil.

The above implementation allows checking whether a node exists or not in O(1).

The other implementations are possible (e.g., using linked lists or hash tables).

Advantages and Disadvantages of Trie

Advantages

- A better search structure than a binary search tree with string keys.
- A more versatile search structure than hash table
- Allows lookup on prefix matching in O(M)-time where M is the length of prefix.
- Allows sorting collection of strings in O(|S|) time where |S| is the total number of characters in all strings

Disadvantages

- On average Tries can be slower (in some cases) than hash tables for looking up patterns/queries.
- Requires a lot of wasteful space, as many nodes, as you descend a trie, will have more and more children set to nil.

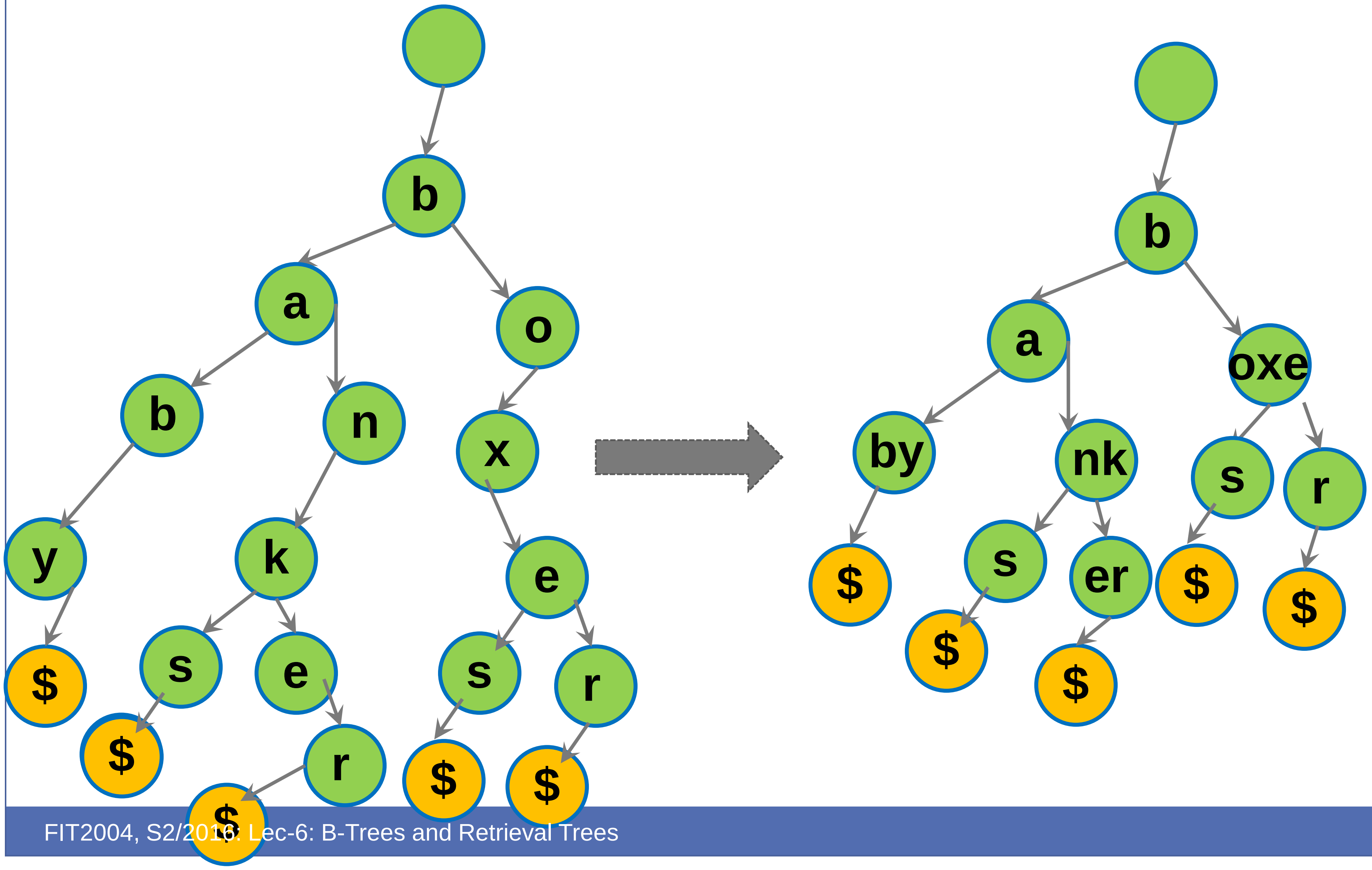
Some properties of Trie

- The maximum depth is the length of longest string in the collection.
- Insertion, Deletion, Lookup operations take time proportional to the length of the string/pattern being inserted, deleted, or searched.
- But, much wasted space with a simple implementation of a Trie, where
  - each node has 1 pointer per symbol in the alphabet.
  - deeper nodes typically have mostly null pointers.
- Can reduce total space usage by turning each node into a linked list or binary search tree etc, trading off time for space.

Radix/PATRICIA Tree (NOT EXAMINABLE BUT WORTH MENTIONING)

- Radix/PATRICIA tree is a space-optimized/compact Trie data structure
- Unlike regular tries, edges can be labeled with substrings of characters.
- The nodes along a path having exactly one child are merged
- This makes them much more efficient for sets of strings that share long prefixes or substrings.

Radix/PATRICIA Tree (NOT EXAMINABLE BUT WORTH MENTIONING)



Suffix Tree

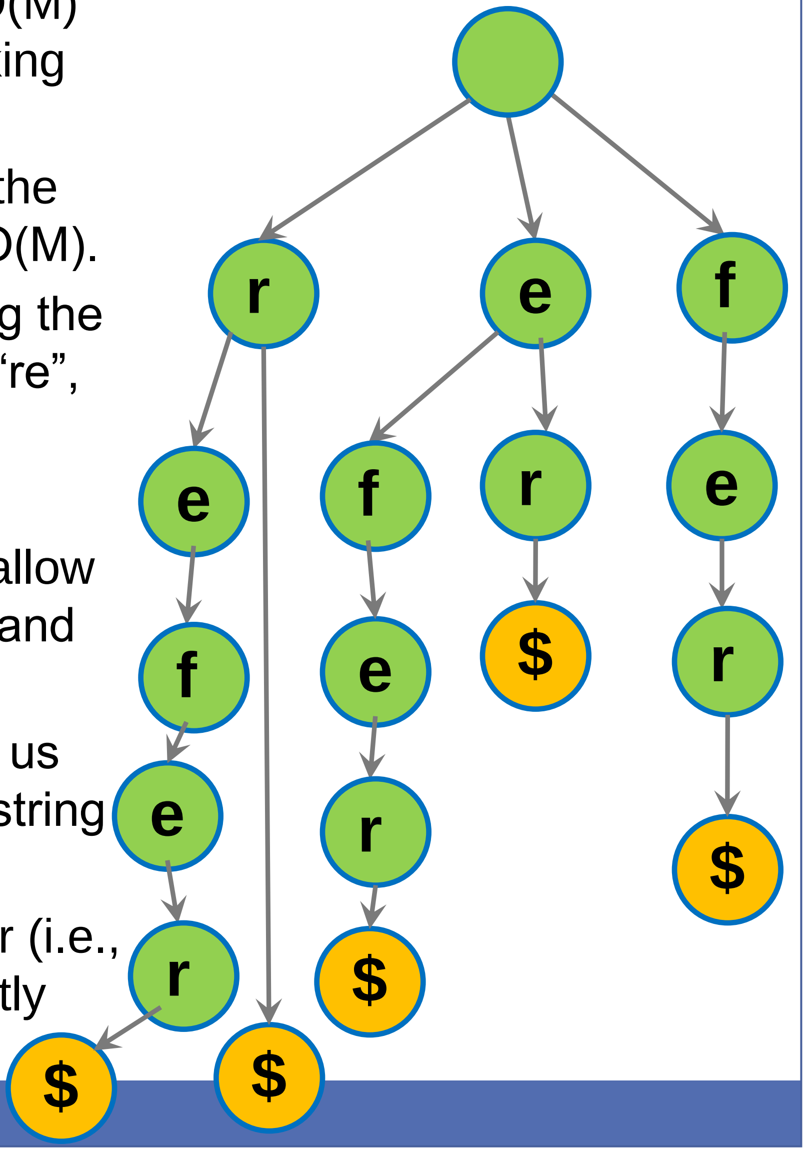
- We saw that a Trie allows prefix matching in O(M) where M is the length of the prefix, e.g., checking whether “ref” is a prefix of “refer”.
- What about substring matching? Can we use the Trie to check if “ef” is a substring of “refer” in O(M).
- No! because “ef” is not a prefix of “refer”. Using the Trie of “refer”, we can only check whether “r”, “re”, “ref”, “refe”, and “refer” are the substrings.

Prefix for a string s[1…M] is a string s[1…X] where X≤M. (e.g., refe is a prefix of refer.)

Suffix for a string s[1…M] is a string s[X…M]. E.g., fer is a suffix of refer.

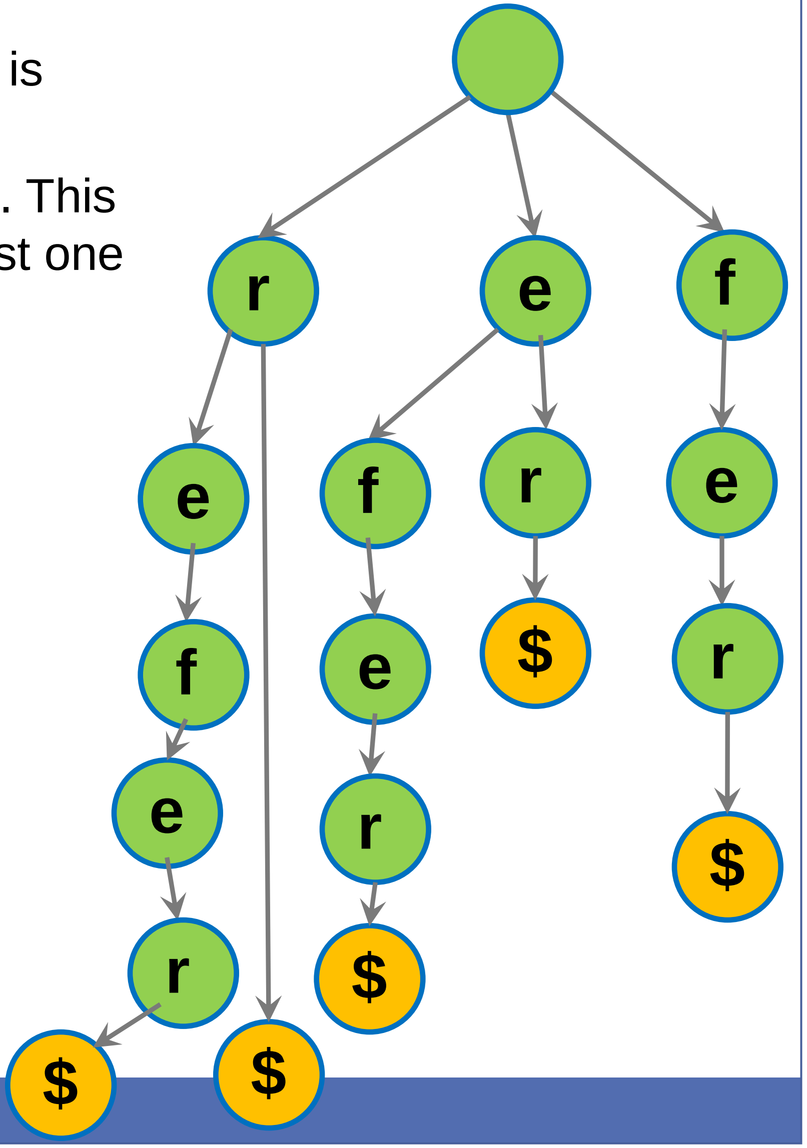
Idea:

- What about if we add “efer” in the Trie? It will allow us efficiently checking whether “e”, “ef”, “efe”, and “efer” are also in the string.
- What about if we also add “fer”. This will allow us checking whether “f”, “fe”, and “fer” are in the string or not.
- In short, if we add all suffixes of the string refer (i.e., refer, efer, fer, er, r) in the Trie, we can efficiently search every substring of refer in the Trie.



Suffix Tree

- Consider some text, e.g., “refer”.
- A Trie constructed using all suffixes of the text is called a Suffix Tree.
- A suffix tree allows efficient substring matches. This is because every substring is a prefix of at least one suffix present in the tree.



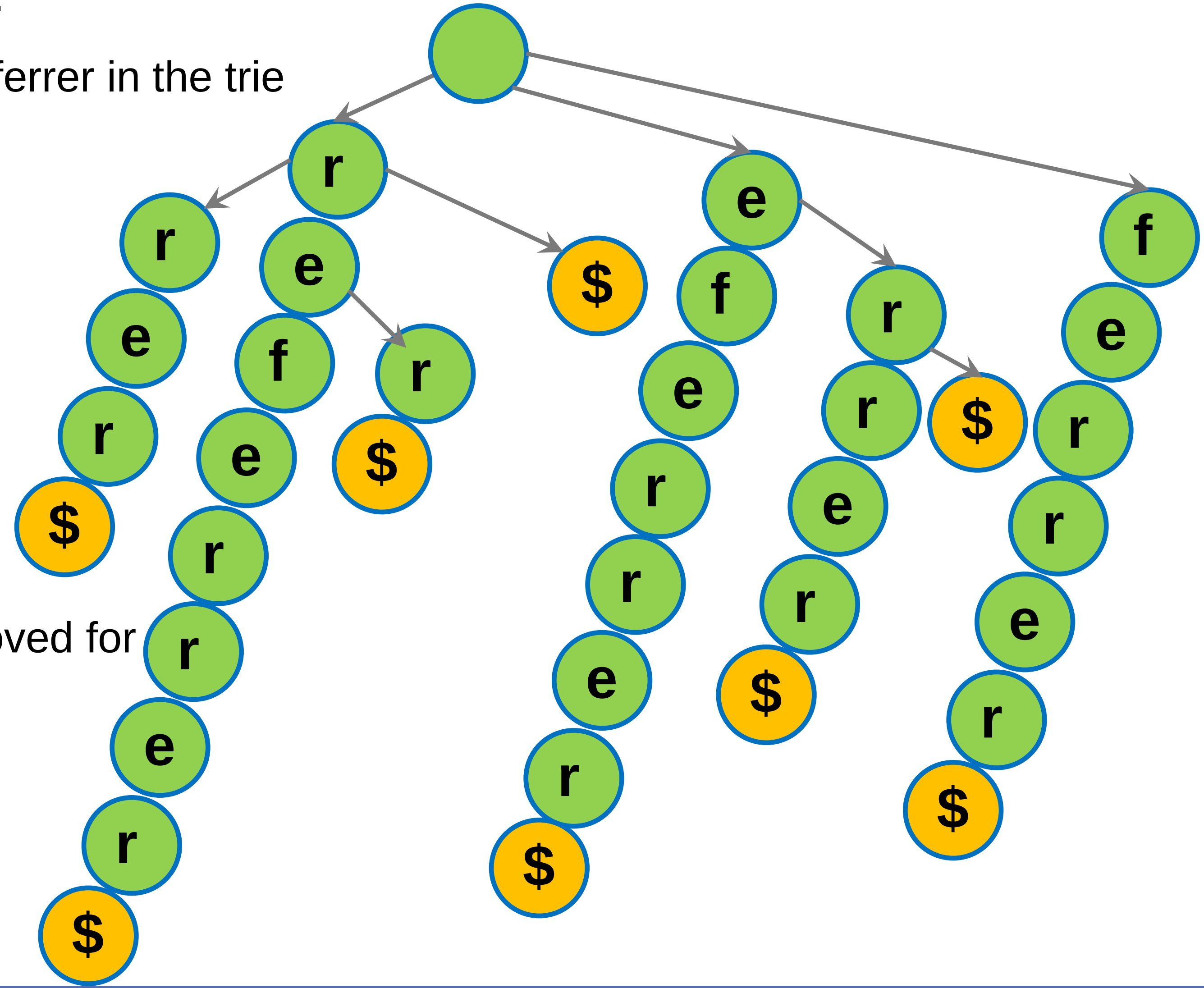
Constructing Suffix Tree

Suffix tree of referrer

Insert all suffixes of referrer in the trie

1. referrer
2. eferrer
3. ferrer
4. errer
5. rrer
6. rer
7. er
8. r

Many arrows are removed for better visualization



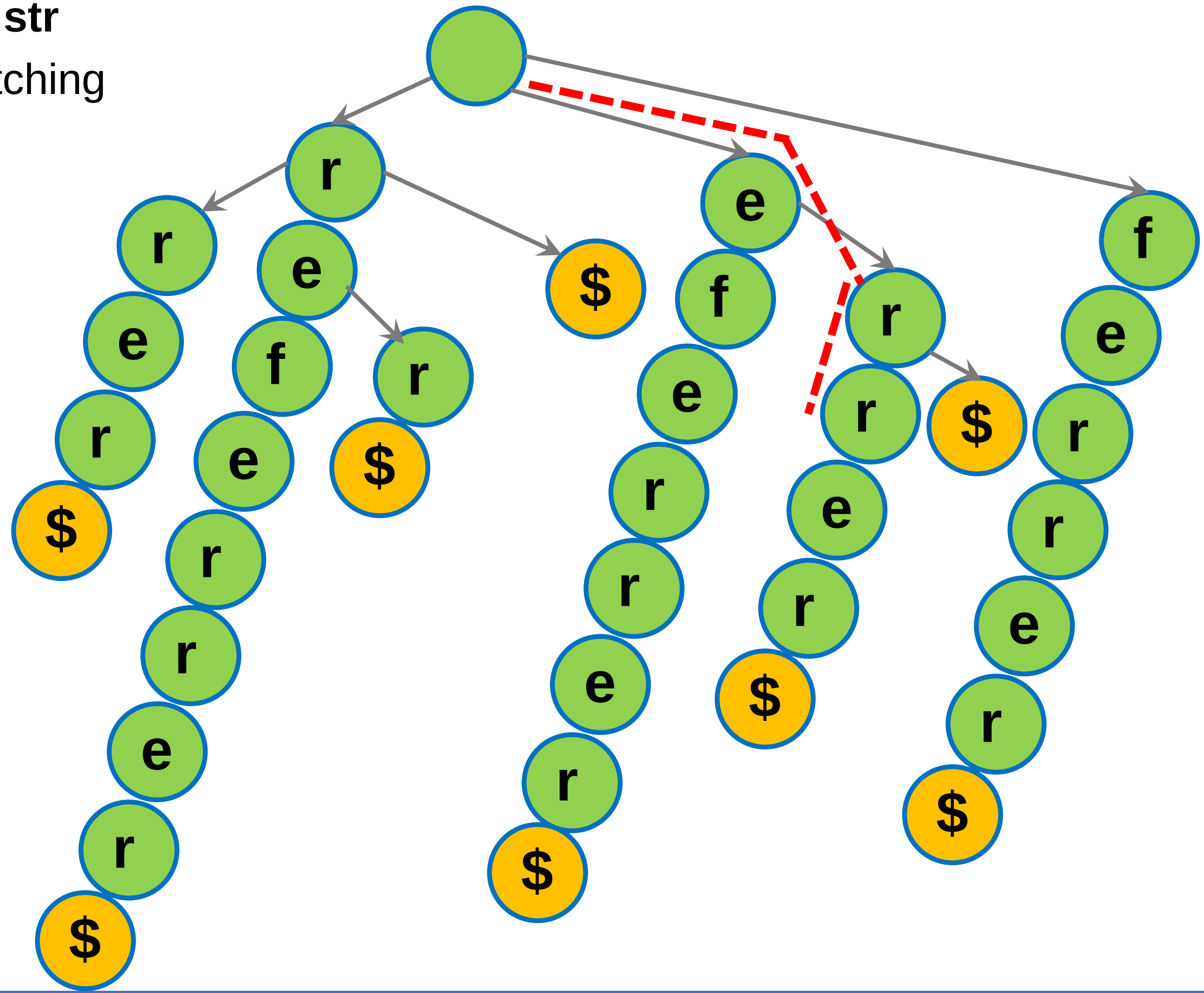
Substring search on Suffix Tree

Substring search for str

- Similar to prefix matching

E.g., search "err"

search "fers"



Time Complexity:

O(M) where M is the length of substring

Counting # of occurences of a substring

- Follow the path similar to prefix matching
- Count # of leaf nodes ($) in the subtree rooted at the last node

E.g., Count “e”

Count “r”

Count “er”

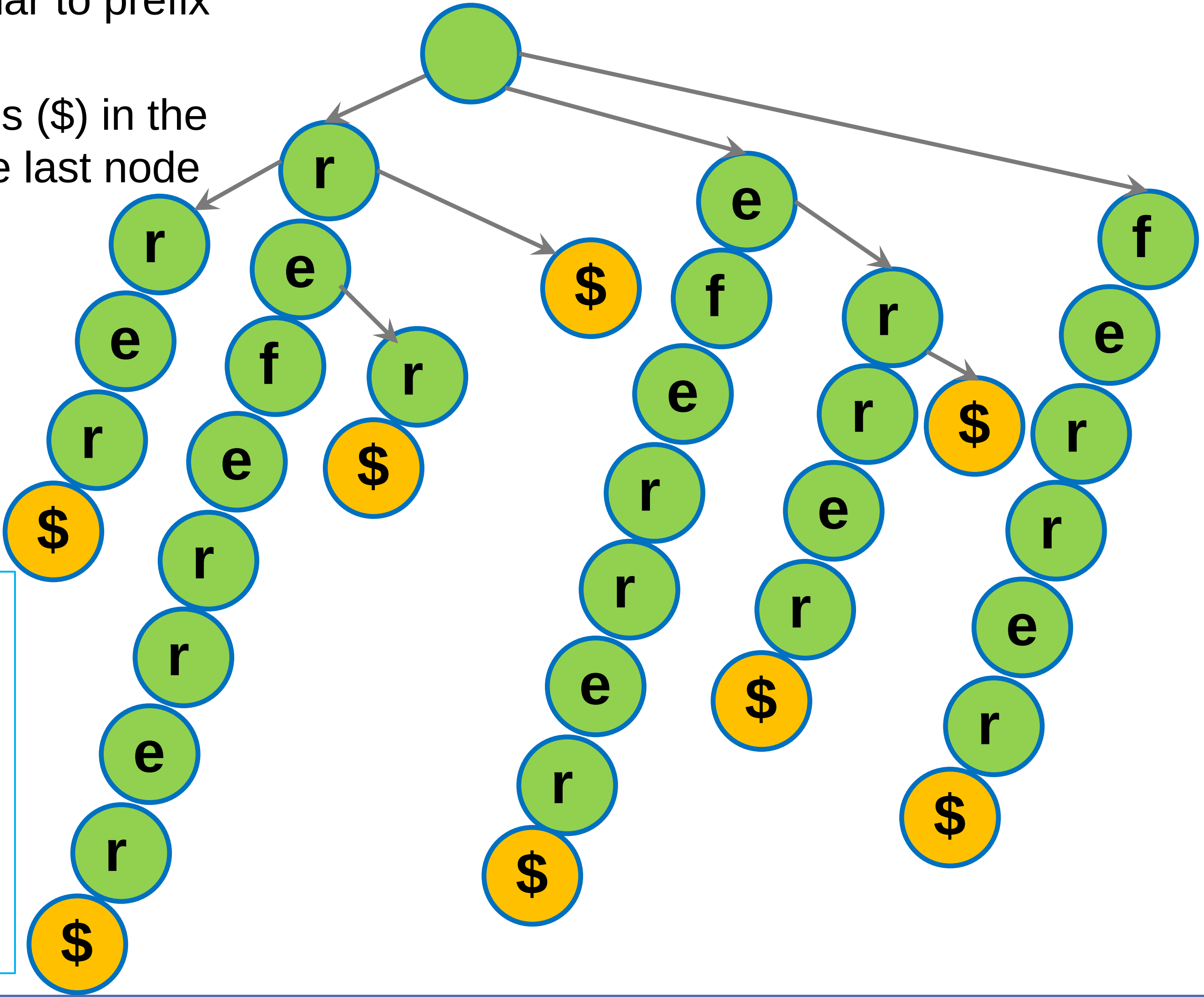
Count “re”

Count “err”

Count “efr”

Time Complexity:

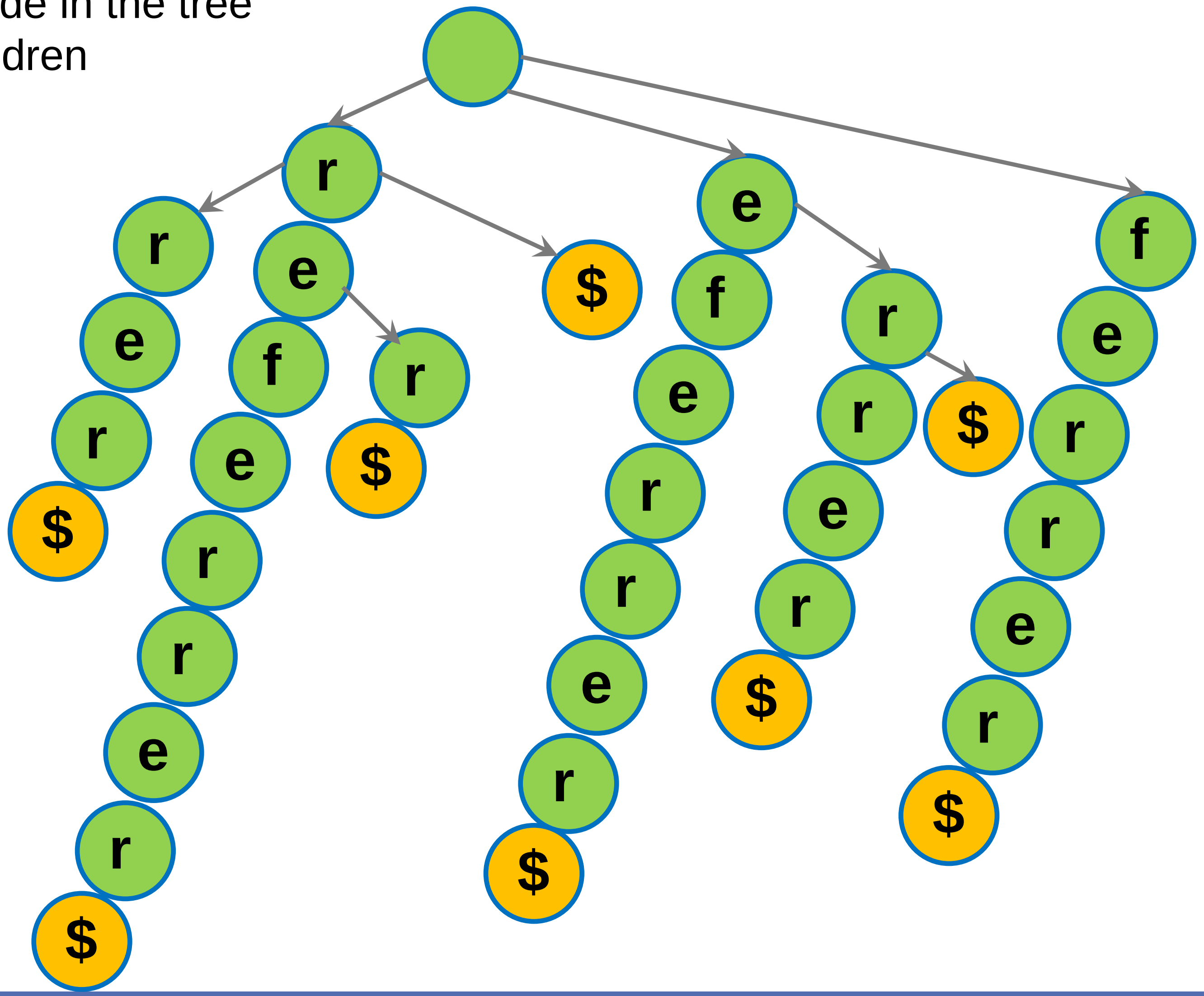
Can be done in O(M) if number of leaf nodes is maintained during construction of suffix tree



Finding longest repeated substring

- Find the deepest node in the tree with at least two children

E.g., “re” and “er”



Space complexity of suffix tree

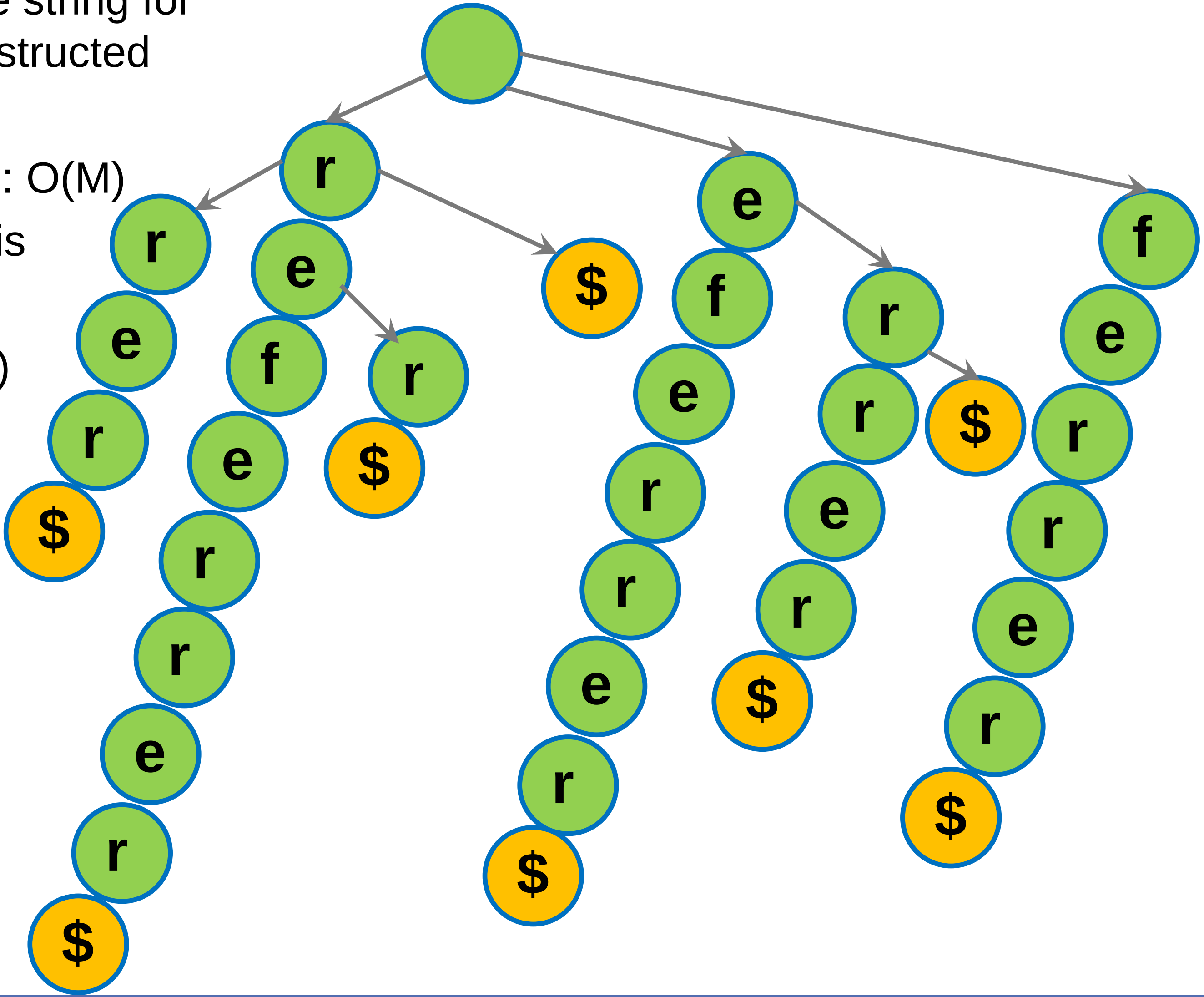
Let M be the size of the string for which suffix tree is constructed

Naïve representation:

- # number of suffixes: O(M)
- Cost for each suffix is

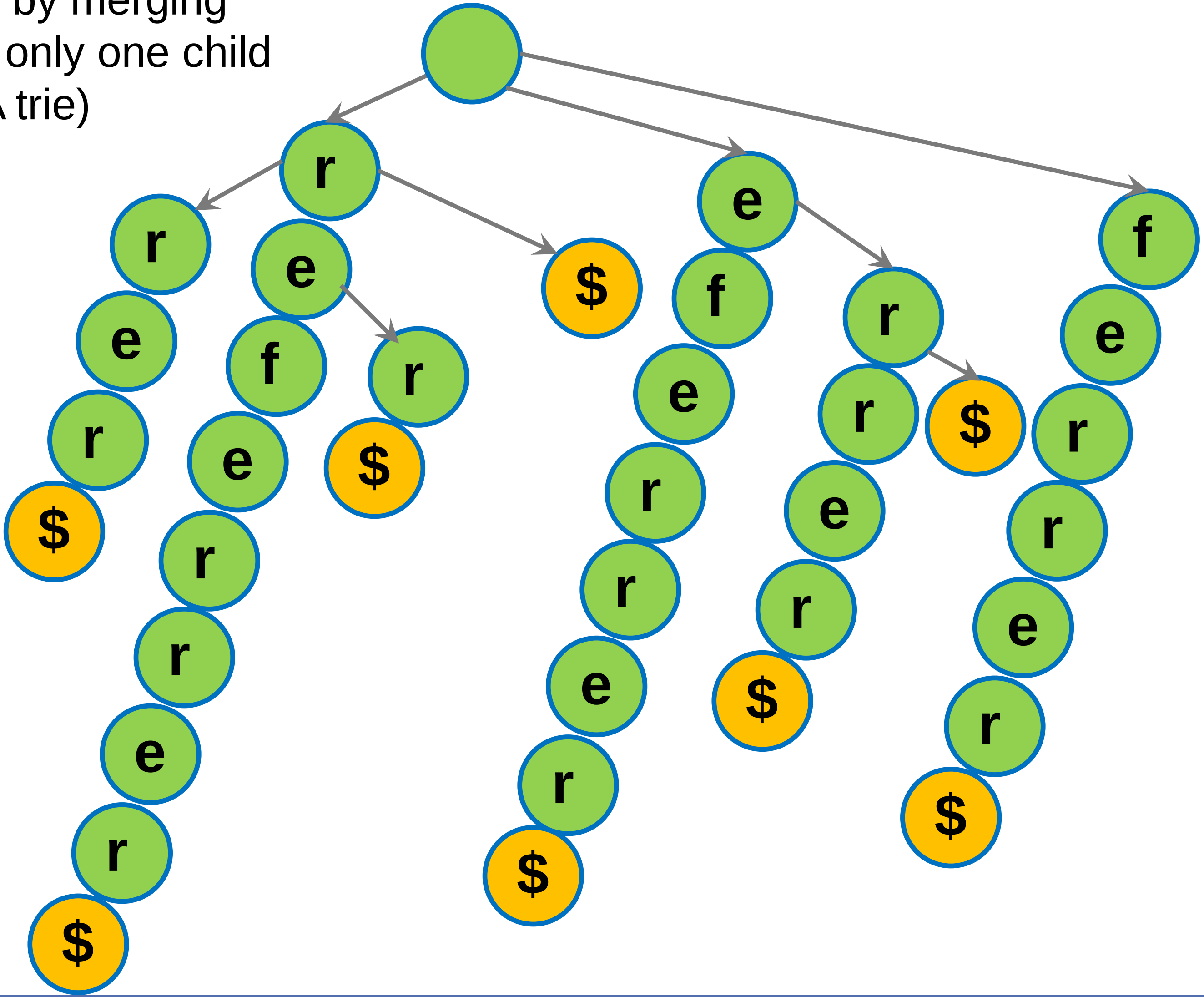
linear to its size

Total space cost: $O(M^2)$



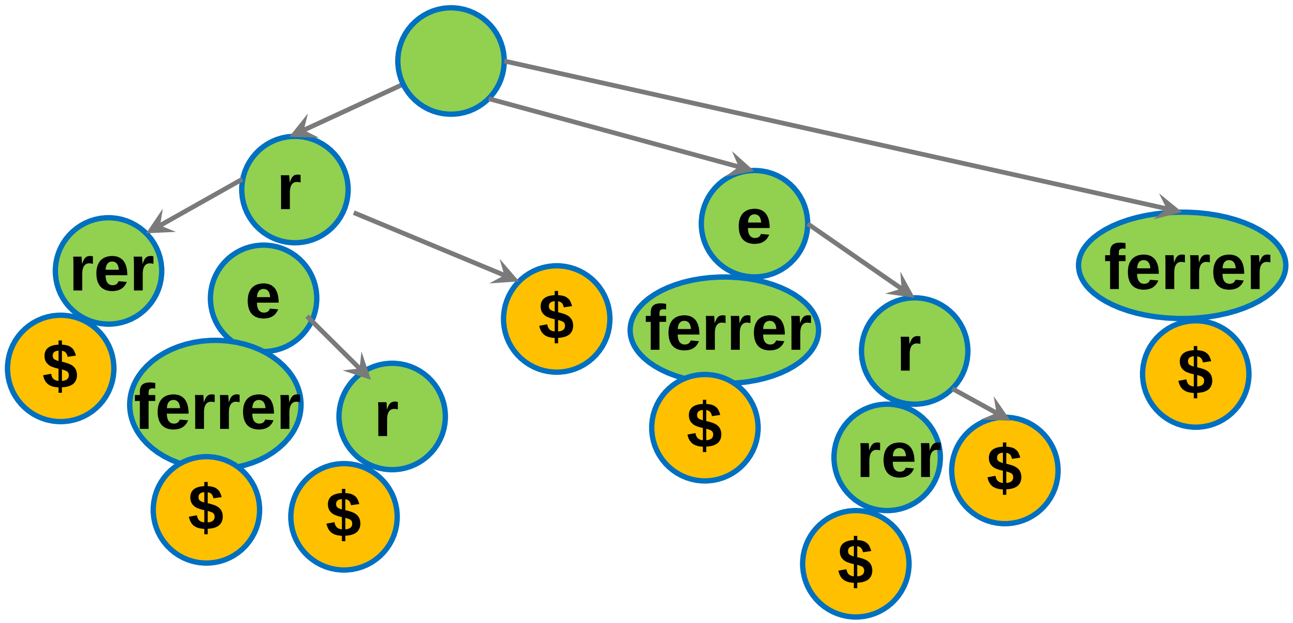
Reducing space complexity of suffix tree

- Compress branches by merging the nodes that have only one child (similar to PATRICIA trie)



Reducing space complexity of suffix tree

- Compress branches by merging the nodes that have only one child (similar to PATRICIA trie)
- But the total complexity is still the same as the same number of letters are stored



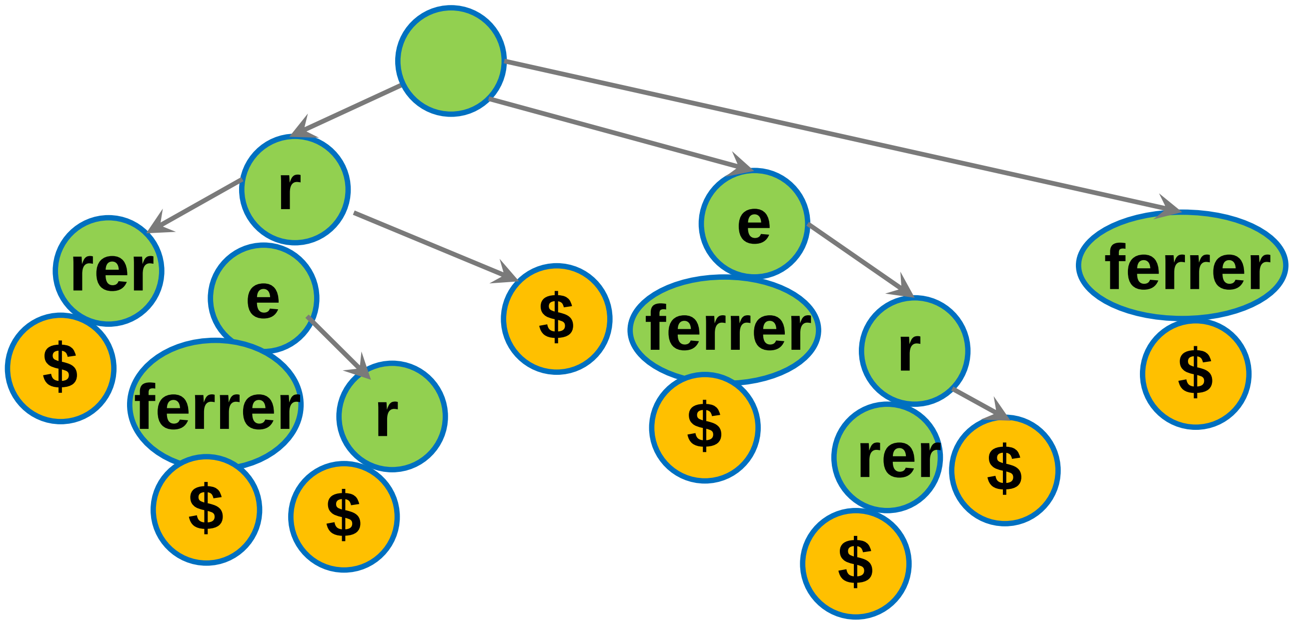
Reducing space complexity of suffix tree

- Compress branches by merging the nodes that have only one child (similar to PATRICIA trie)
- But the total complexity is still the same as the same number of letters are stored
- Replace every substring with numbers (x,y) where x is the starting index of the substring and y is its length

e.g., ferrer is represented as (3,6)

rer is represented as (6,3)

r	e	f	e	r	r	e	r
1	2	3	4	5	6	7	8



Reducing space complexity of suffix tree

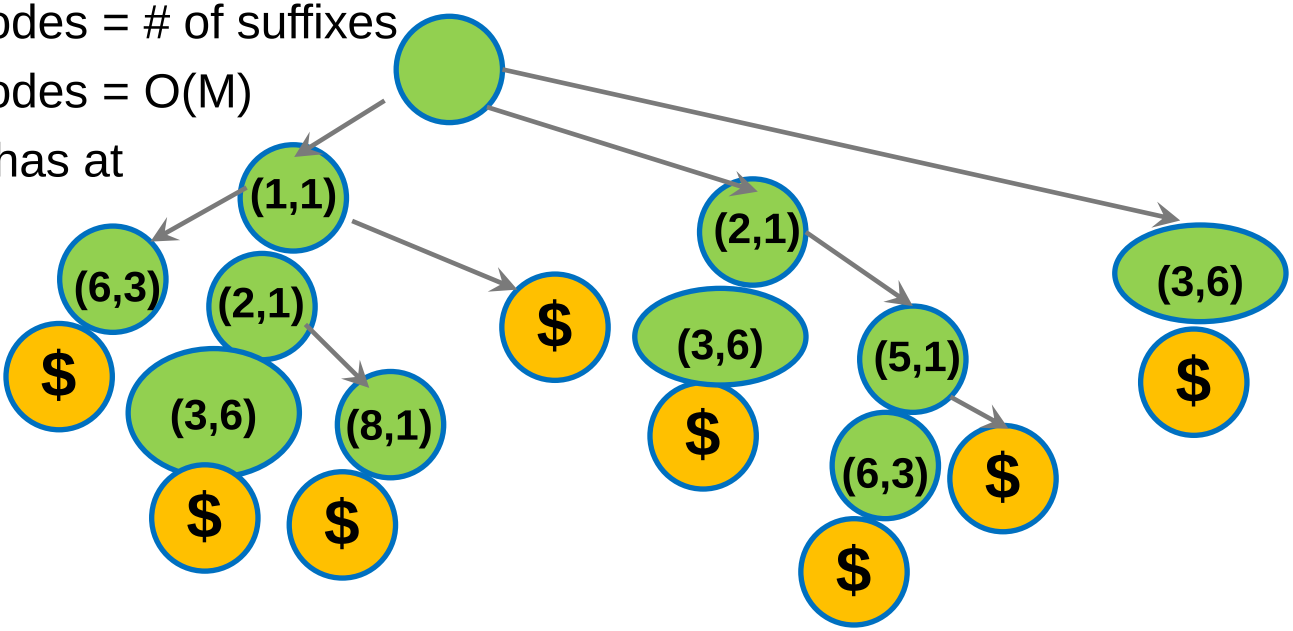
- Compress branches by merging the nodes that have only one child (similar to PATRICIA trie)
- But the total complexity is still the same as the same number of letters are stored
- Replace every substring with numbers (x,y) where x is the starting index of the substring and y is its length

  e.g., ferrer is represented as (3,6)

  rer is represented as (6,3)

r	e	f	e	r	r	e	r
1	2	3	4	5	6	7	8

- Total number of leaf nodes = # of suffixes
- Total number of leaf nodes = O(M)
- Each node in the tree has at least two children
- So, total # of nodes is O(M + M/2 + M/4 + …) = O(M)



Time complexity of constructing suffix tree

- The algorithm described earlier inserts O(M) suffixes
- Insertion cost of each suffix is linear to the size of suffix
- Thus, total time complexity is $O(M^2)$

It is possible to construct suffix tree in O(M)

- Esko Ukkonen in 1995 gave a beautiful (but involved) algorithm to construct a Suffix Tree in linear time. If you every get interested in doing this in linear time, consider reading the source:

Ukkonen, E. (1995). "On-line construction of sux trees". Algorithmica 14 (3): 249260.

Summary

Take home message

- B-tree provides an efficient search structure on disk resident data
- Trie and Suffix trees provide efficient text search and pattern matching (typically linear in number of characters in string)

Things to do (this list is not exhaustive)

- Read more about B-tree and make sure you understand how insertion, search and deletion etc. work on B-tree
- Implement Trie and Suffix trees and run various pattern matching queries

Coming Up Next

- Burrows-Wheeler Transform - A beautiful space-time efficient pattern matching algorithm on text