



University of
Pittsburgh

Algorithms and Data Structures 2

CS 1501



Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
 - Homework 7: next Friday @ 11:59 pm
 - Lab 6: Monday 10/31 @ 11:59 pm
 - Nothing due this week
- Midterm Exam
 - Wednesday 10/19 (MW Section) and Thursday 10/20 (TuTh Section)
 - in-person, closed-book
- Live QA Session on Piazza every Friday 4:30-5:30 pm
- Video for debugging using VS Code

Previous lecture

- LZW
 - implementation concerns
- Shannon's Entropy
- Comparing LZW vs Huffman
- Burrows-Wheeler Compression Algorithm
- ADT Priority Queue
 - array and BST implementations

This Lecture

- ADT Priority Queue (PQ)
 - Heap implementation
- Heap Sort
- Indexable PQ

Repetitive Highest Priority Problem

- Input:

- a (large) dynamic set of data items
 - each item has a priority
 - e.g., highest priority is minimum item
 - e.g., highest priority is maximum item
- a *stream* of zero or more of each of the following operations
 - Find a highest priority item in the set
 - Insert an item to the set
 - Remove a highest priority item from the set

- Examples

- Selection sort
 - Repeatedly, remove a minimum item from the array and insert it in its correct position in the sorted array
- Huffman trie construction
 - Each iteration: remove a minimum tree from the forest (**twice**) and insert a new tree

Let's create an ADT!

- The ADT Priority Queue (PQ)

- Primary operations of the PQ:

- Insert

- Find item with highest priority

- e.g., findMin() or findMax()


- Remove an item with highest priority

- e.g., removeMin() or removeMax()

Is a BST overkill to implement ADT PQ?

- Balanced BST (e.g., RB-BST) provides $\log n$ runtime time for all operations
- Our find and remove operations only need the highest priority item, not to find/remove *any* item
 - Can we take advantage of this to improve our runtime?
 - Yes!

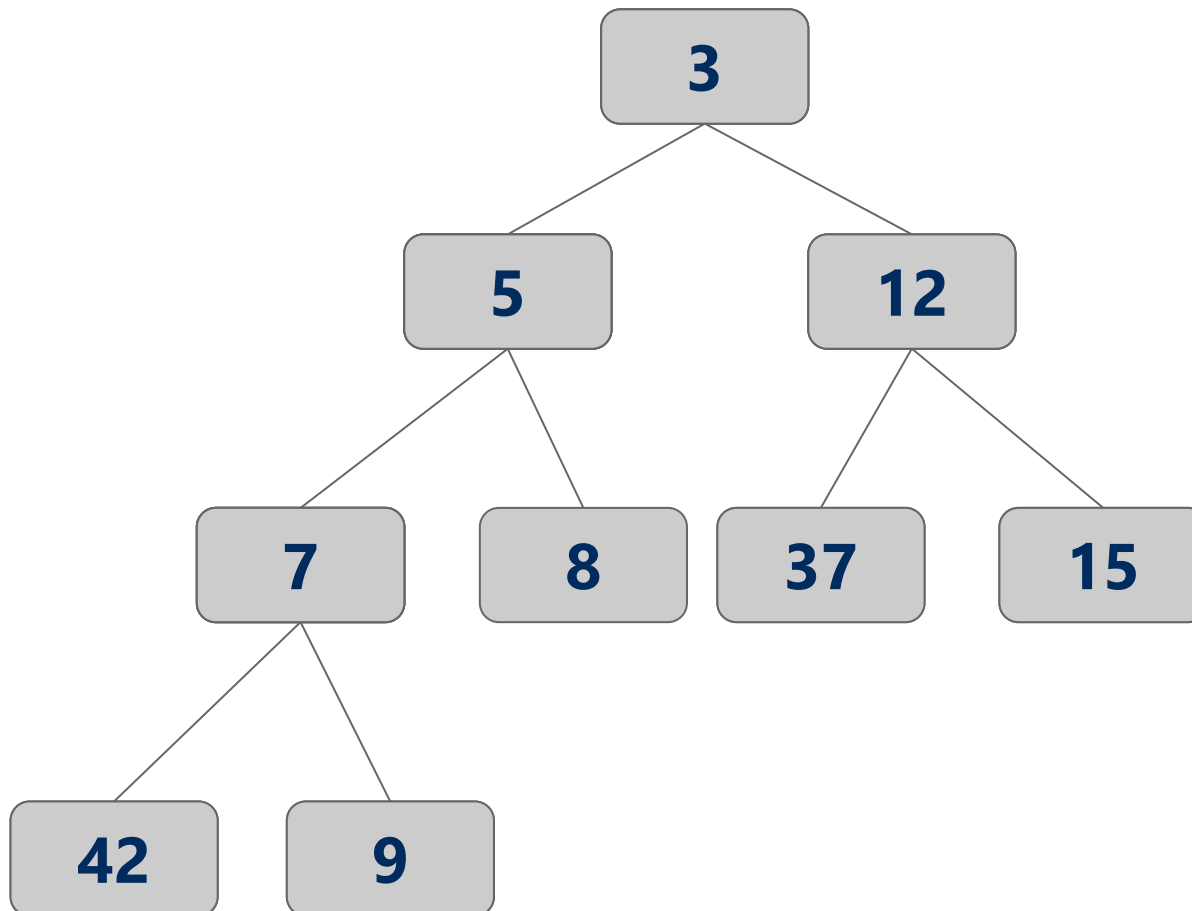
The heap

- 
- A heap is **complete** binary tree such that for each node T in the tree:
 - T.item is of a higher priority than T.right_child.item
 - T.item is of a higher priority than T.left_child.item
 - It does not matter how T.left_child.item relates to T.right_child.item
 - This is a relaxation of the approach needed by a BST

The heap property

Min Heap Example

- In a Min Heap, a highest priority item is a minimum item



Heap PQ runtimes

- Find is easy
 - Simply the root of the tree
 - $\Theta(1)$
- Remove and insert are not quite so trivial
 - The tree is modified and the heap property must be maintained

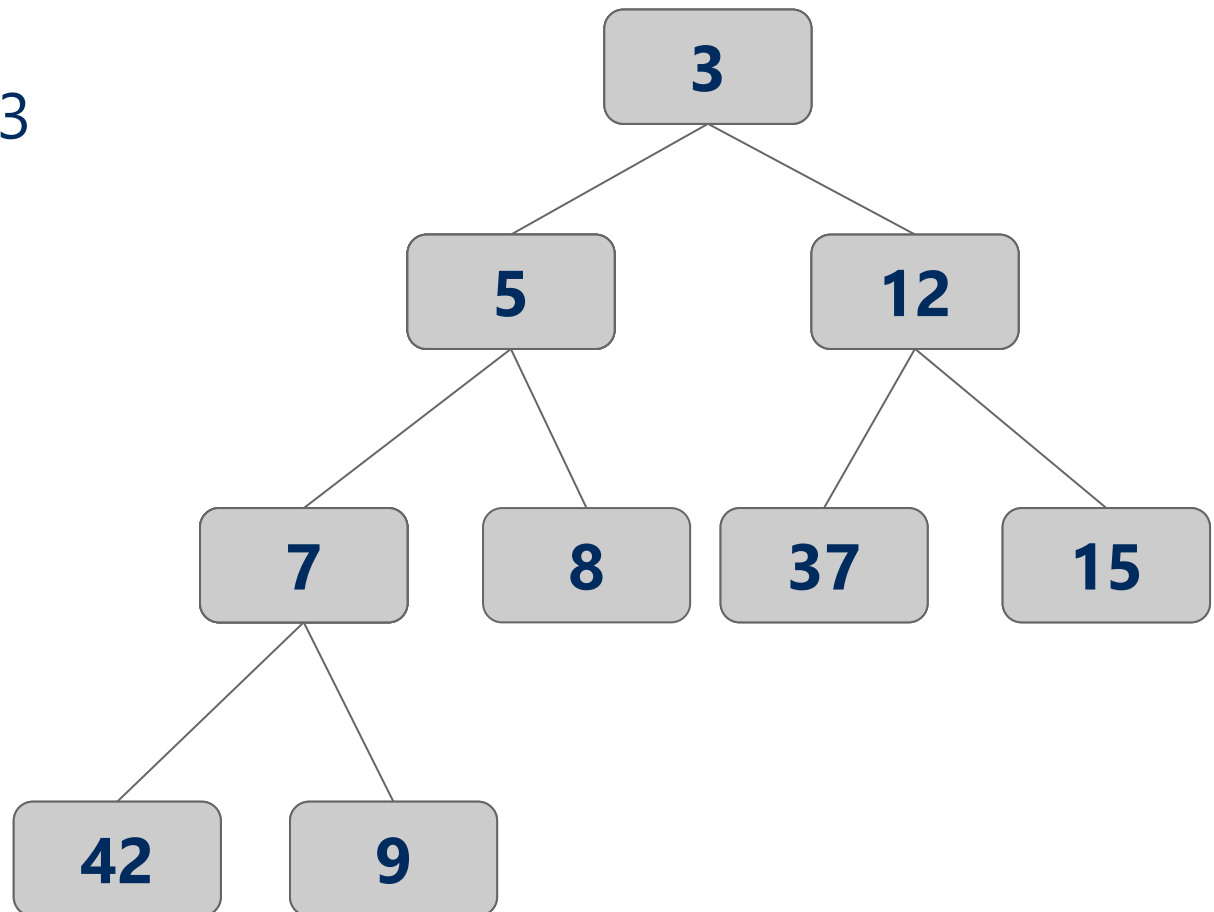
Heap insert

- Add a new node at the next available leaf
- Push the new node up the tree until it is supporting the heap property

Min heap insert

Insert:

7, 42, 37, 5, 8, 15, 12, 9, 3

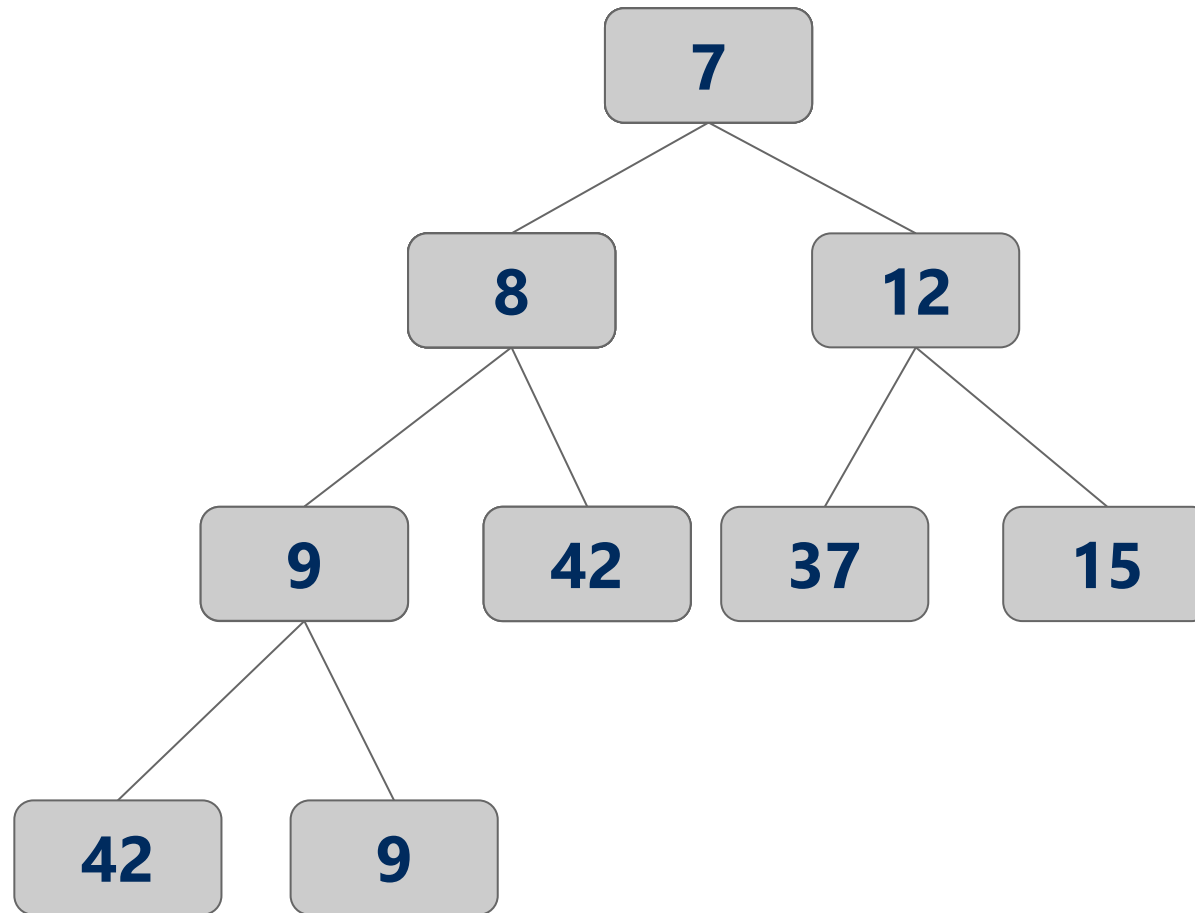


Heap remove

- Tricky to delete root...
 - So let's simply overwrite the root with the item from the last leaf and delete the last leaf
 - But then the root is violating the heap property...
 - So we push the root down the tree until it is supporting the heap property

Min heap removal

NO!



Heap runtimes

- Find
 - $\Theta(1)$
- Insert and remove
 - Height of a complete binary tree is $\lg n$
 - At most, upheap and downheap operations traverse the height of the tree
 - Hence, insert and remove are $\Theta(\lg n)$

Heap implementation

- Simply implement tree nodes like for BST
 - This requires overhead for dynamic node allocation
 - Also must follow chains of parent/child relations to traverse the tree
- Note that a heap will be a complete binary tree...
 - We can easily represent a complete binary tree using an array

Storing a heap in an array

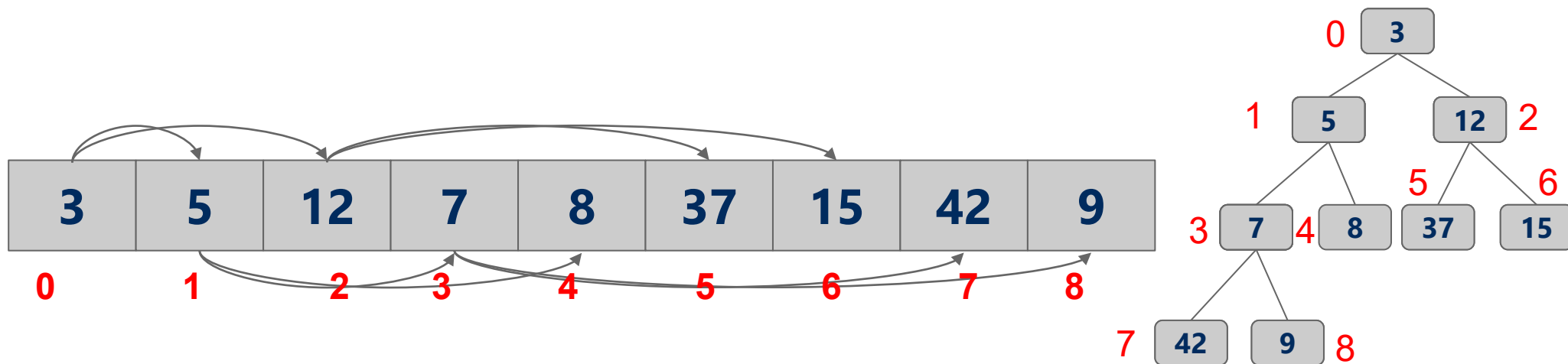
- Number nodes row-wise starting at 0
- Use these numbers as indices in the array
- Now, for node at index i

- $\text{parent}(i) = \lfloor (i - 1) / 2 \rfloor$

- $\text{left_child}(i) = 2i + 1$

- $\text{right_child}(i) = 2i + 2$

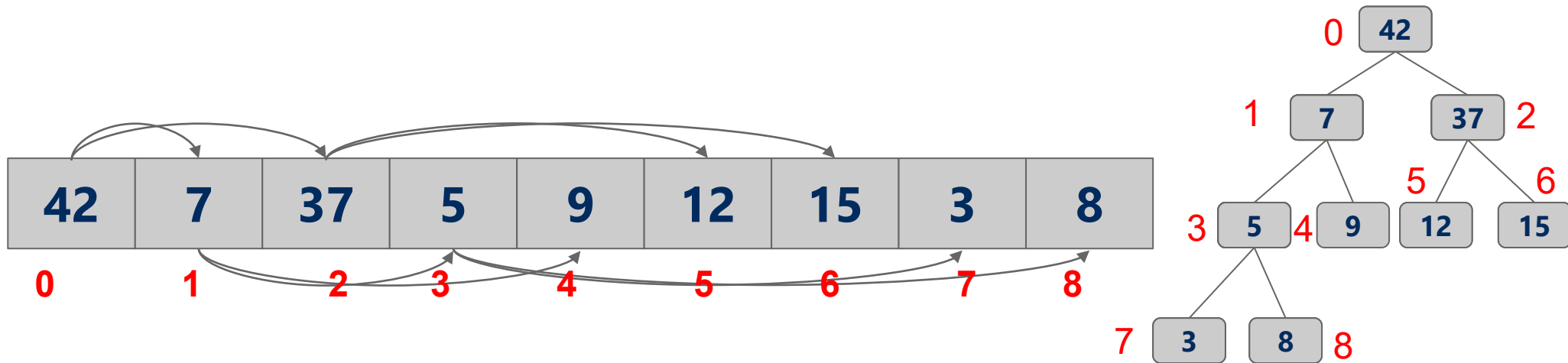
For arrays indexed from 0



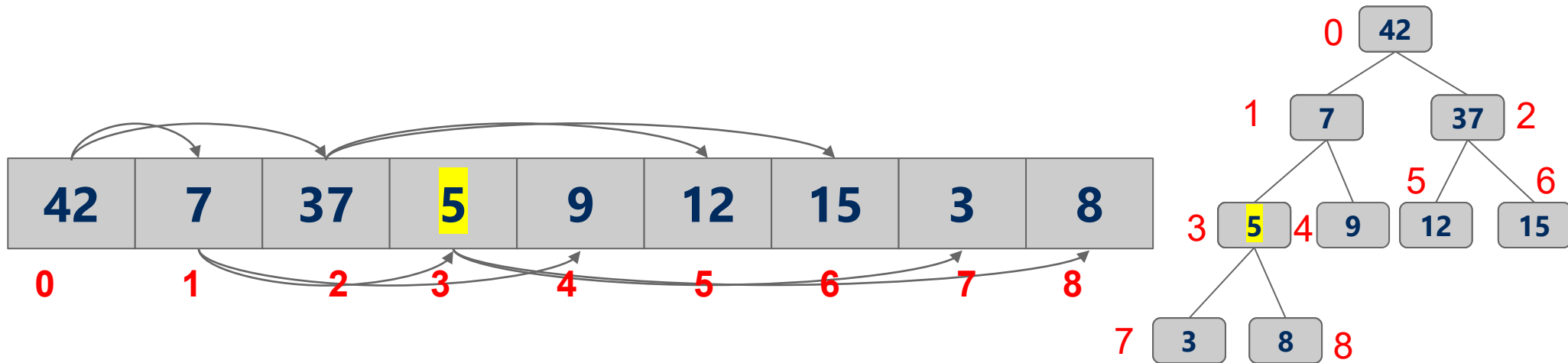
Can we turn any array into a heap?

- Yes!
- Any array can be thought of as a complete tree!
- We can change it into a heap using the following algorithm
- Scan through the array **right to left** starting from the rightmost non-leaf
 - the largest index i such that $\text{left_child}(i)$ is a valid index (i.e., $< n$)
 - $2i+1 < n \rightarrow i < (n-1)/2$
 - push the node down the tree until it is supporting the heap property
- This is called the **Heapify** operation

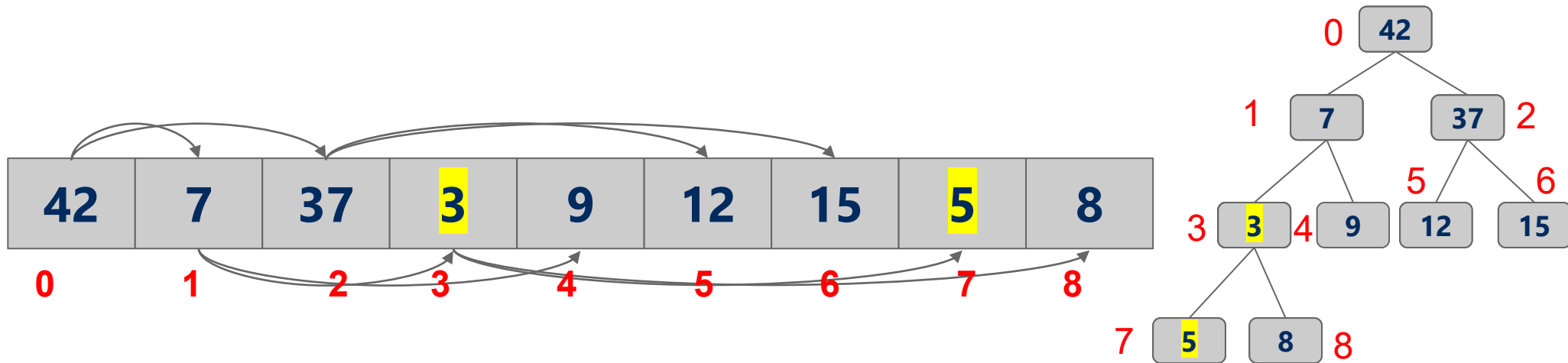
Heapify Example: Building a Min Heap



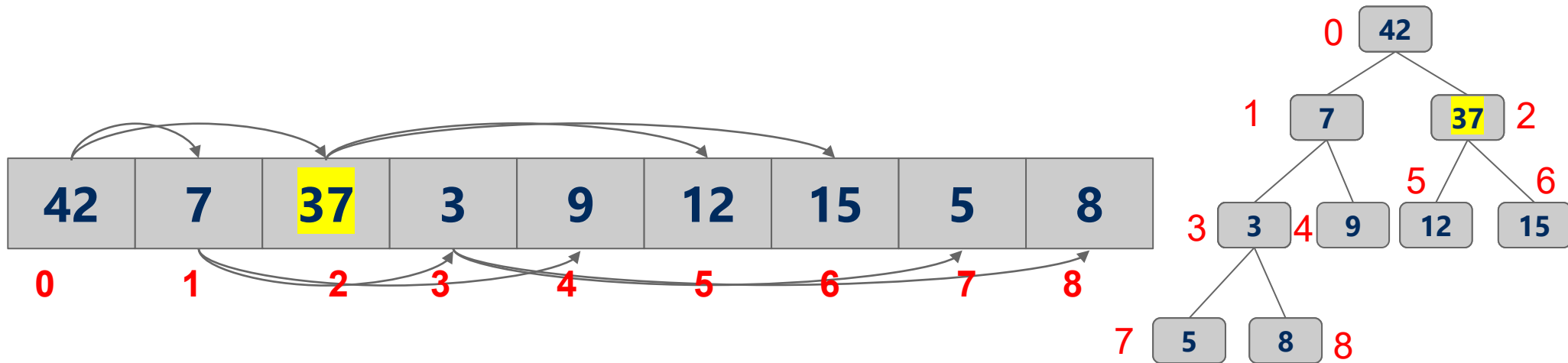
Heapify Example: Building a Min Heap



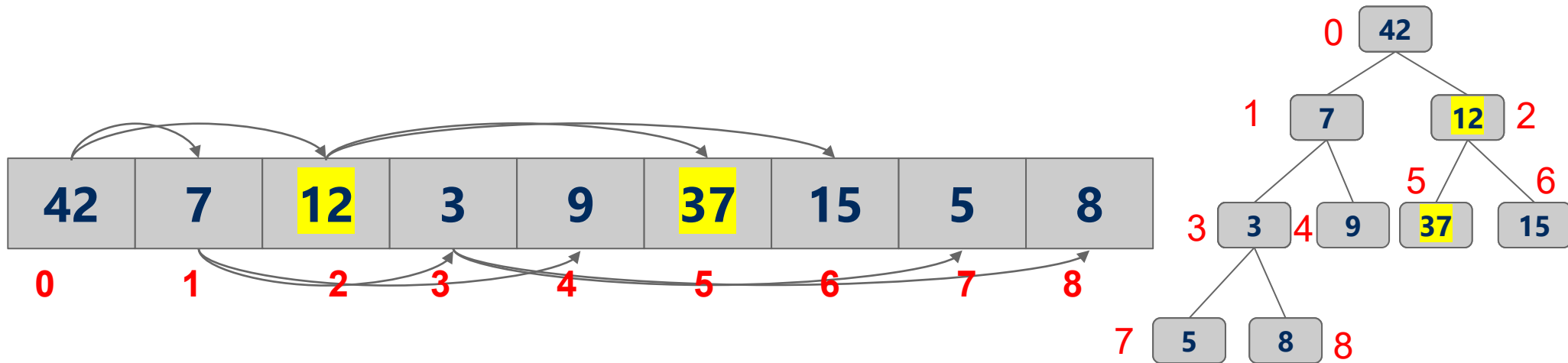
Heapify Example: Building a Min Heap



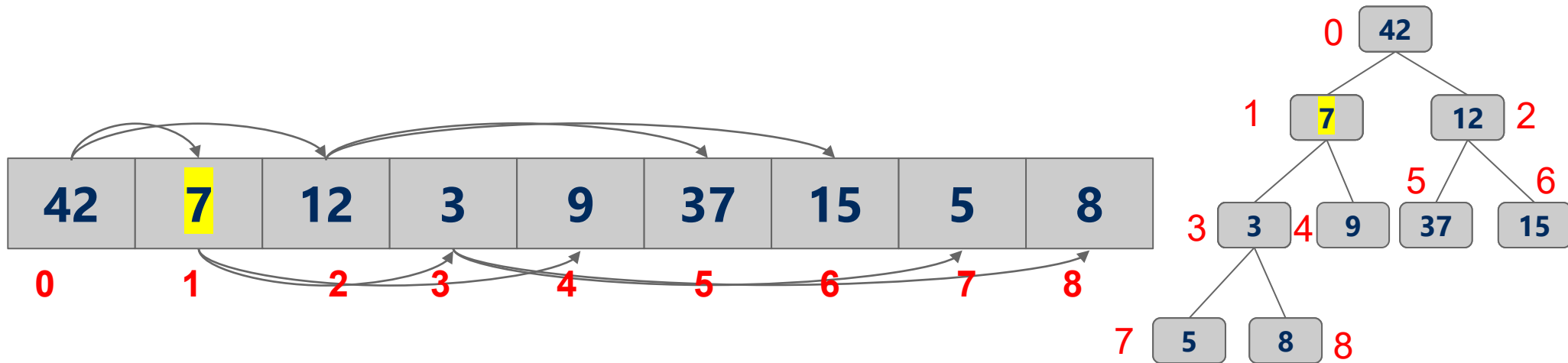
Heapify Example: Building a Min Heap



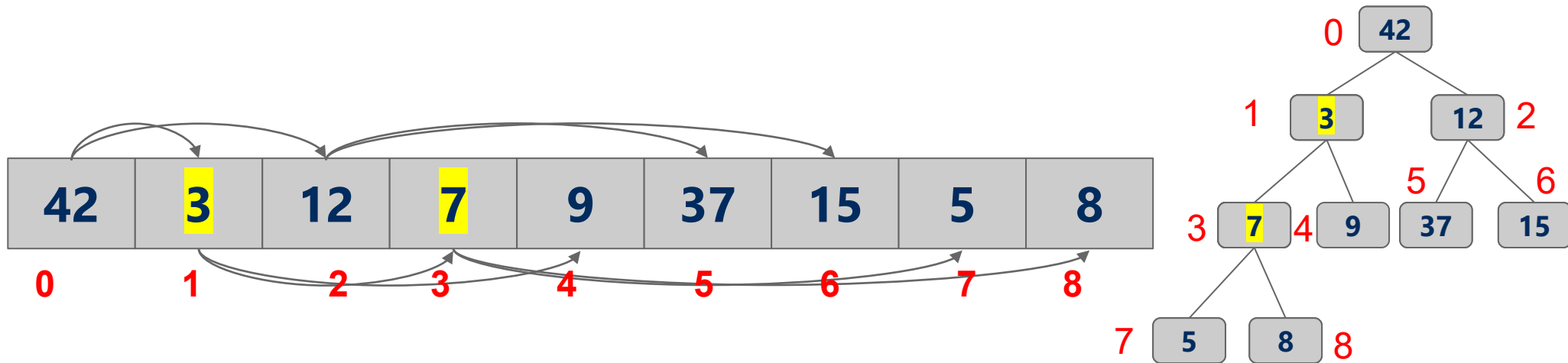
Heapify Example: Building a Min Heap



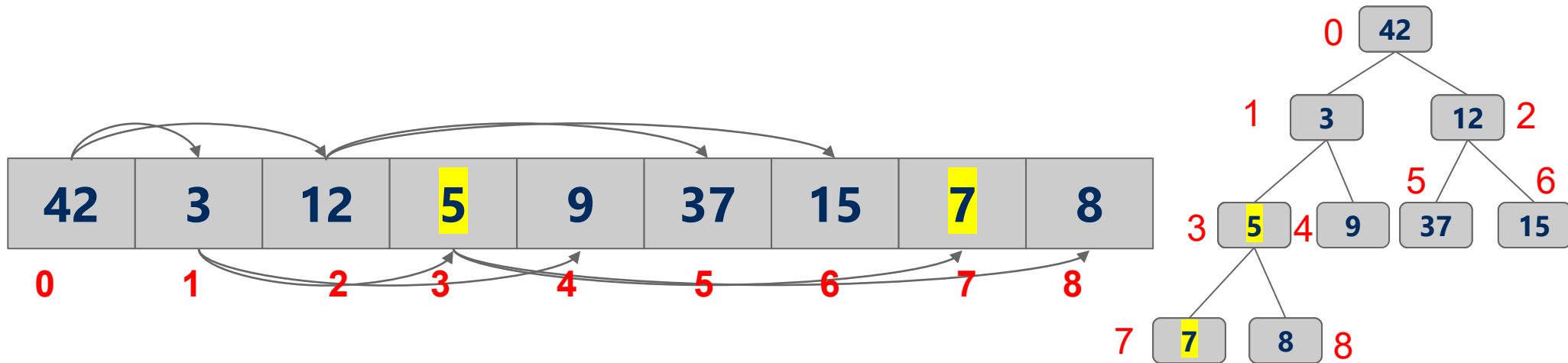
Heapify Example: Building a Min Heap



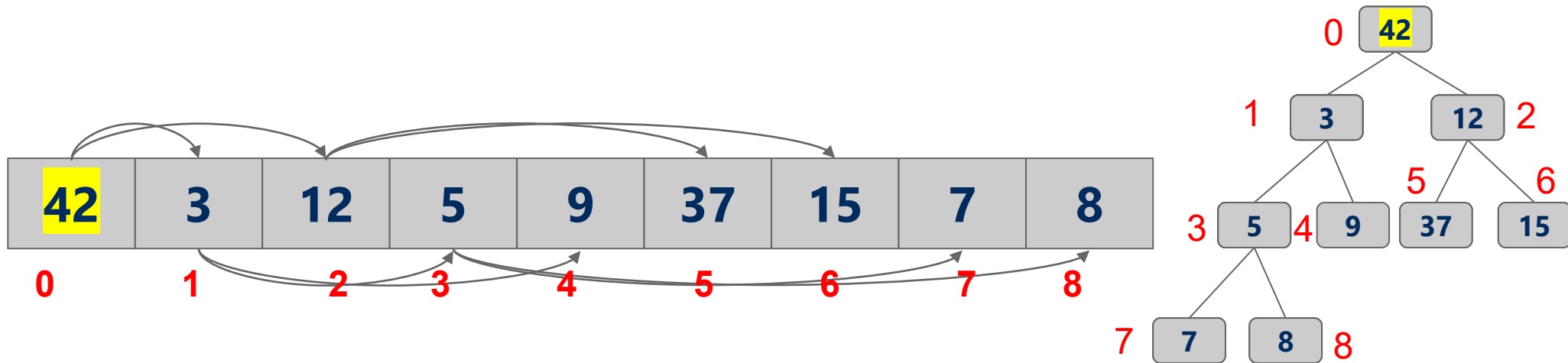
Heapify Example: Building a Min Heap



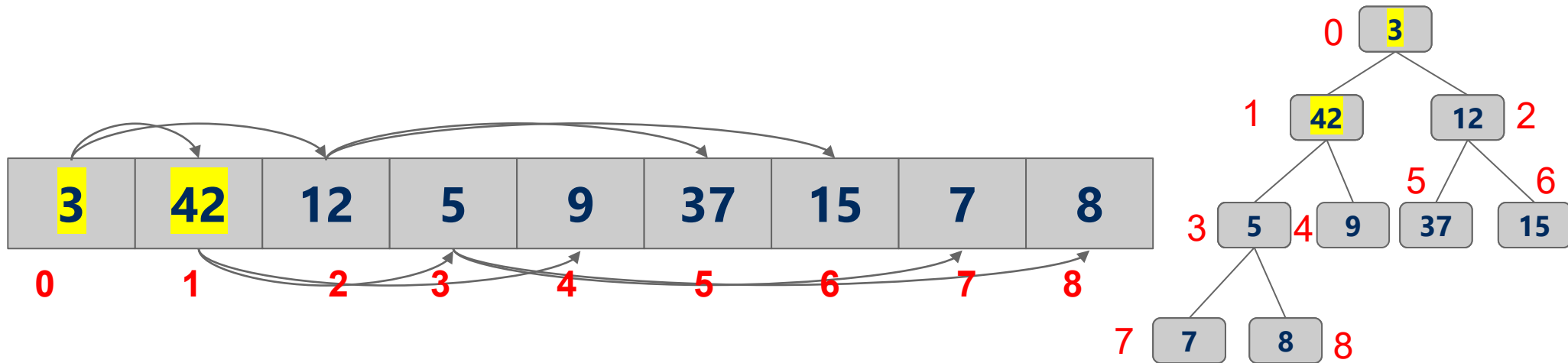
Heapify Example: Building a Min Heap



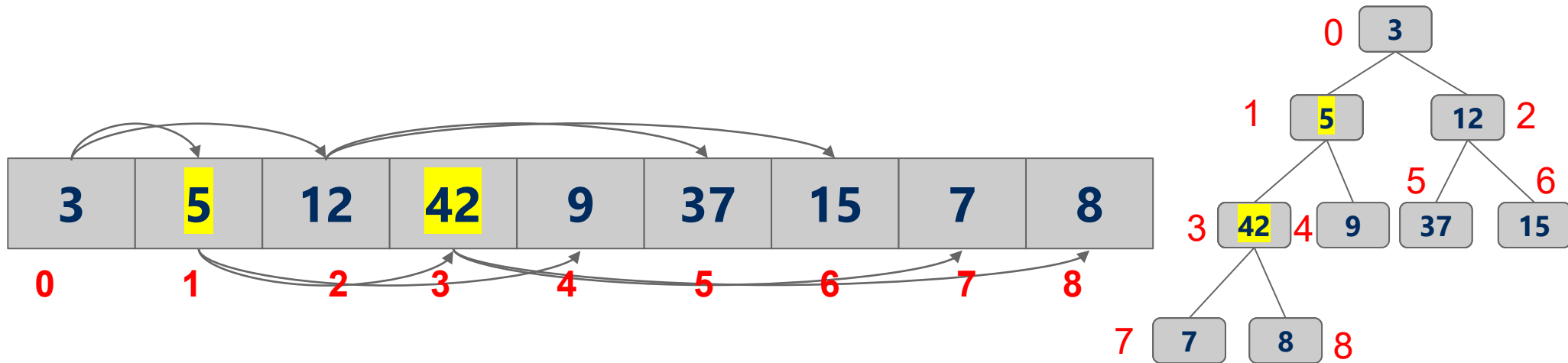
Heapify Example: Building a Min Heap



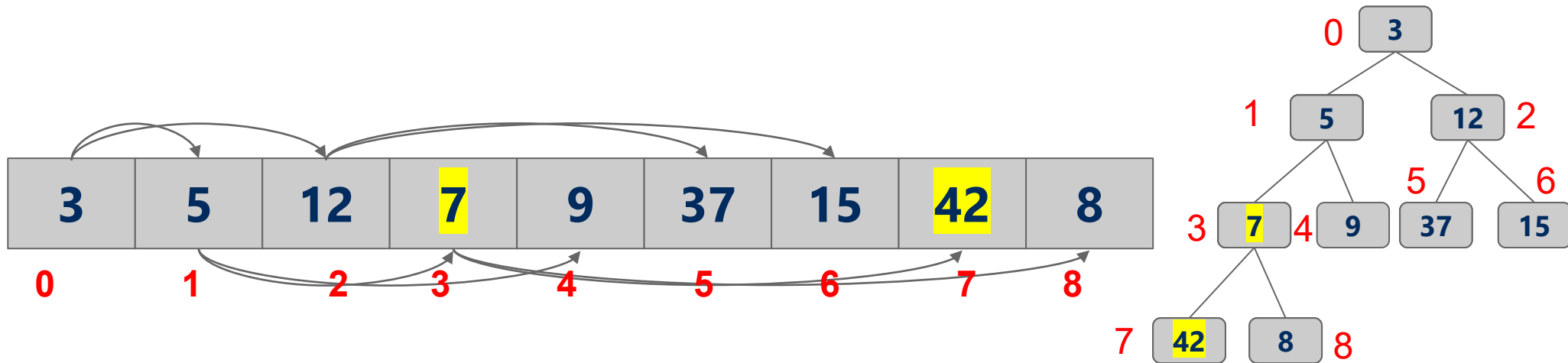
Heapify Example: Building a Min Heap



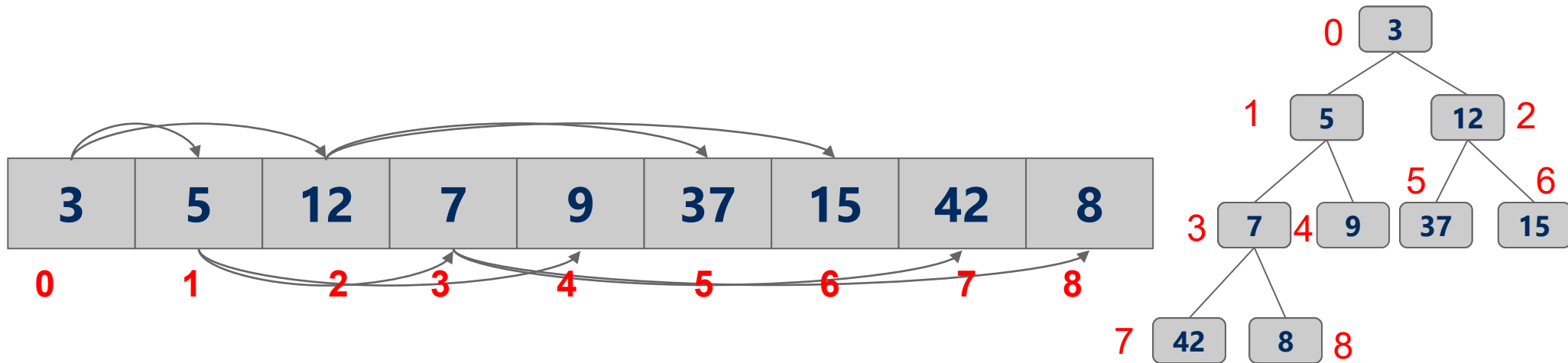
Heapify Example: Building a Min Heap



Heapify Example: Building a Min Heap



Heapify Example: Building a Min Heap

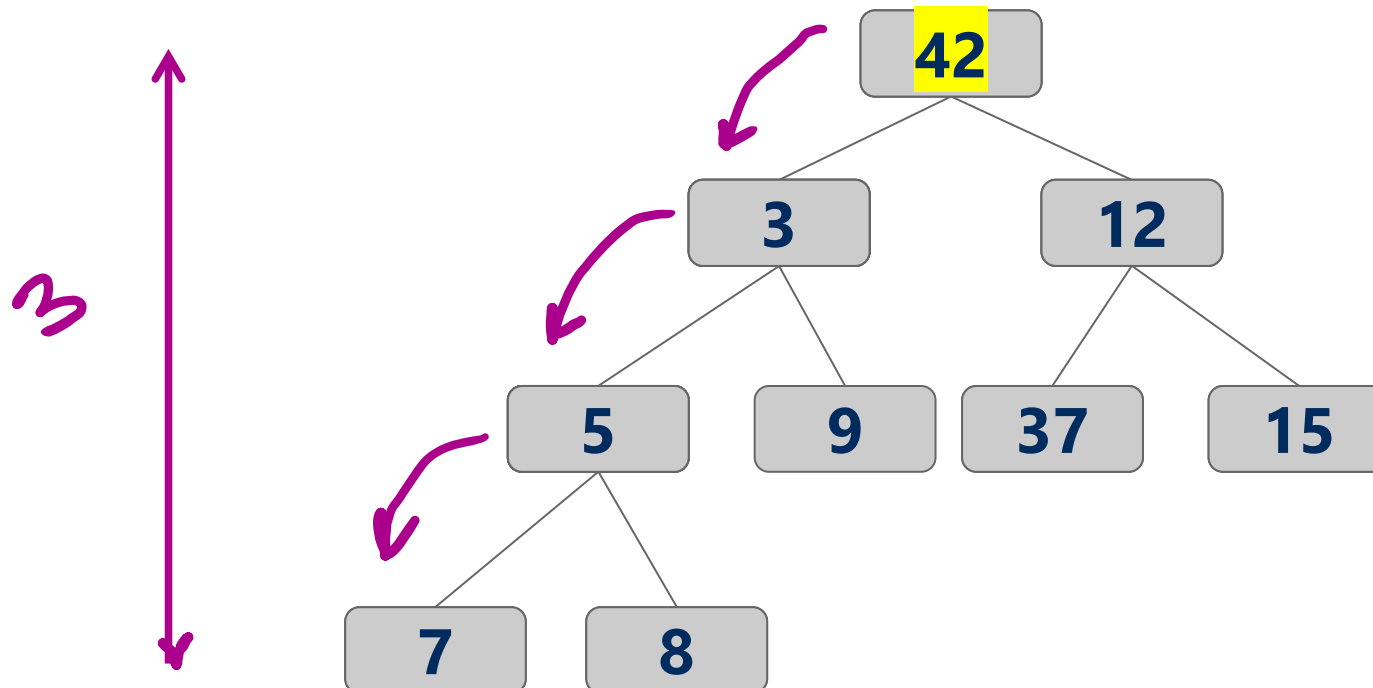


Heapify Running time

- Upper bound analysis:
 - We make about $n/2$ downheap operations
 - $\log n$ each
 - So, $O(n \log n)$

Heapify Running time

- A tighter analysis
 - for each node that we start from, we make at most $height[node]$ swaps

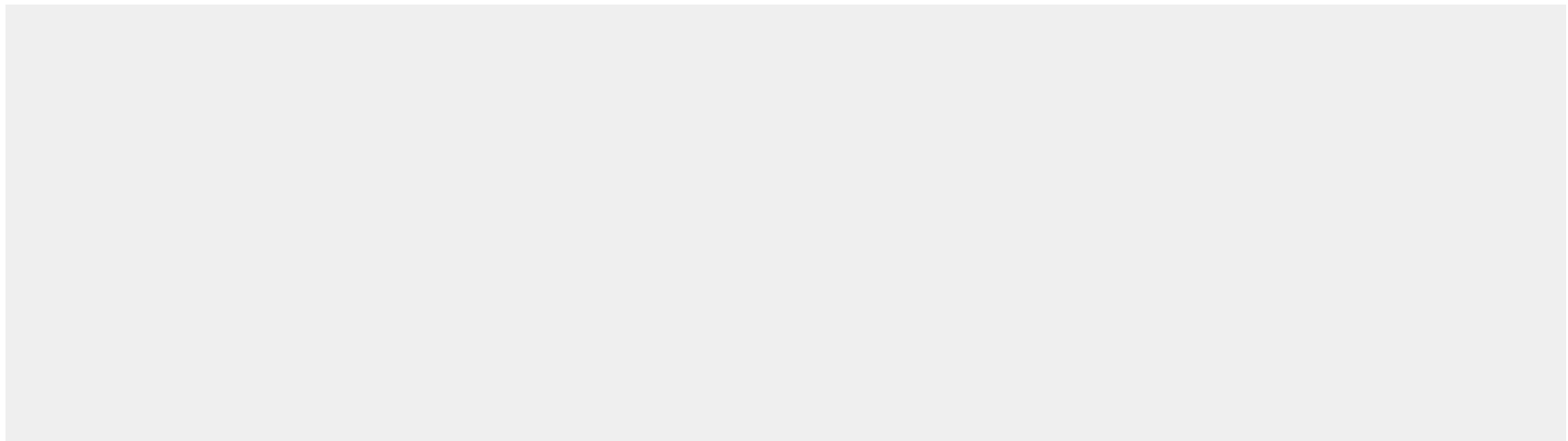


Heapify Running time: A tighter analysis

- $Runtime = \sum_{i=1}^n height[n]$
- $= \sum_{i=0}^{\log n} \text{number of nodes with height } i$
- Assume a full tree
 - A node with height i has 2^i nodes in its subtree including itself
 - Assume k nodes with height i :
 - they will have $k2^i$ nodes in their subtrees
 - $k2^i \leq n \rightarrow k \leq n/2^i$
- So, at most $n/2^i$ nodes exist with height i
- $\sum_{i=0}^{\log n} \frac{n}{2^i} = n + \frac{n}{2} + \frac{n}{4} + \dots$
- $= \theta(\text{largest term}) = \theta(n)$

Heap Sort

- Heapify the numbers
 - MAX heap to sort ascending
 - MIN heap to sort descending
- "Remove" the root
 - Don't actually delete the leaf node
- Consider the heap to be from 0 .. length - 1
- Repeat



Heap sort analysis

- Runtime:
 - Worst case:
 - $n \log n$
- In-place?
 - Yes
- Stable?
 - No

Storing Objects in PQ

- What if we want to update an Object in the heap?
 - What is the runtime to find an arbitrary item in a heap?
 - $\Theta(n)$
 - Hence, updating an item in the heap is $\Theta(n)$
 - Can we improve of this?
 - Back the PQ with something other than a heap?
 - Develop a clever workaround?

Indirection

- Maintain a second data structure that maps item IDs to each item's current position in the heap
- This creates an *indexable* PQ

Indirection example setup

- Let's say I'm shopping for a new video card and want to build a heap to help me keep track of the lowest price available from different stores.
- Keep objects of the following type in the heap:

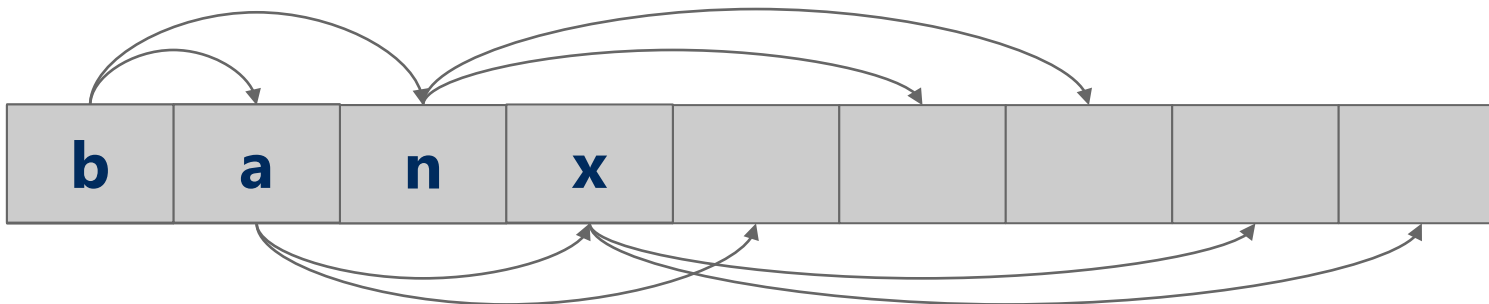
```
class CardPrice implements Comparable<CardPrice>{  
    public String store;  
    public double price;  
    public CardPrice(String s, double p) { ... }  
    public int compareTo(CardPrice o) {  
        if (price < o.price) { return -1; }  
        else if (price > o.price) { return 1; }  
        else { return 0; }  
    }  
}
```

Indirection example

- `n = new CardPrice("NE", 333.98);`
 - `a = new CardPrice("AMZN", 339.99);`
 - `x = new CardPrice("NCIX", 338.00);`
 - `b = new CardPrice("BB", 349.99);`
-
- Update price for NE: 340.00
 - Update price for NCIX: 345.00
 - Update price for BB: 200.00

Indirection

"NE":2
"AMZN":1
"NCIX":3
"BB":0



Indexable PQ Discussion

- How are our runtimes affected?
- space utilization?
- how should we implement the indirection?
- what are the tradeoffs?

Muddiest Points

- **Q: what is going to be on the midterm?**
- Up to and including material covered on Monday 10/10 and Tuesday 10/11
- Please check the study guide on Canvas
 - practice test on GradeScope
 - old exam
 - will try to post the answer as soon as possible

Muddiest Points

- **Q: I'm confused about entropy and the equations for that. How is it useful?**
- It helps us determine the “information content” in a file.
- In lossless compression, a file cannot be compressed to less than its entropy

Muddiest Points

- **Q: How do we determine the entropy of a given file we're trying to compress?**
- Given that
 - the file has n total characters and K unique characters,
 - $f(c)$ is the frequency of character c in the file
- Shannon's entropy: $H(\text{file}) = \sum_{c=0}^K \frac{f(c)}{n} \log_2\left(\frac{f(c)}{n}\right)$ bits/character
- Underlying assumption:
 - the file has been generated by a source that produces **independent characters**
 - **Huffman Compression is optimal under that assumption**
 - This assumption may be wrong though!
 - repeated long strings \rightarrow use LZW
 - long sequences of identical characters \rightarrow use RLE (Run Length Encoding)
 - We may need to try different compression algorithms on the file

Muddiest Points

- **Q: What might be some examples in where we might pick one compression type over the other?**
- Depends on the structure of the input file
- long strings of identical values
 - → RLE
- repeated long strings
 - → LZW
- frequently occurring values
 - → Huffman
- few different characters
 - → fixed-length codewords with < 8 bits

Muddiest Points

- **Q: Does entropy play a role in the implementation of the compression algorithm itself? Or is it only used for selecting the best algorithm?**
- Some compression algorithms attempt to reach Shannon's Entropy lower bound on the file size
 - e.g., Huffman Encoding and Arithmetic Encoding

Muddiest Points

- **Q: PQ runtimes for different data structures**

	findMin	removeMin	insert
Unsorted Array	$O(n)$	$O(n)$	$O(1)$
Sorted Array	$O(1)$	$O(1)$	$O(n)$
Red-Black BST	$O(\log n)$	$O(\log n)$	$O(\log n)$

Muddiest Points

- **Q: PQ runtimes for different data structures**

	findMin	removeMin	insert
Unsorted Array	$O(n)$	$O(n)$	$O(1)$
Sorted Array	$O(1)$	$O(1)$	$O(n)$
Red-Black BST	$O(\log n)$	$O(\log n)$	$O(\log n)$

Muddiest Points

- **Q: Not exactly clear on move to front encoding**

Input:

e a e d e e

0

a

1

b

2

c

3

d

4

e

Output:

Muddiest Points

- **Q: Not exactly clear on move to front encoding**

Input:

e a e d e e

0

a

1

b

2

c

3

d

4

e

Output:

4

Muddiest Points

- **Q: Not exactly clear on move to front encoding**

Input:

e a e d e e

0

a

e

1

b

a

2

c

b

3

d

c

4

e

d

Output:

4

Muddiest Points

- **Q: Not exactly clear on move to front encoding**

Input:

e a e d e e

0

a

e

a

1

b

a

e

2

c

b

b

3

d

c

c

4

e

d

d

Output:

4

1

Muddiest Points

- **Q: Not exactly clear on move to front encoding**

Input:

e a e d e e

0	a	e	a	e
1	b	a	e	a
2	c	b	b	b
3	d	c	c	c
4	e	d	d	d

Output:

4 1 1

Muddiest Points

- **Q: Not exactly clear on move to front encoding**

Input:

e a e d e e

0	a	e	a	e	d
1	b	a	e	a	e
2	c	b	b	b	a
3	d	c	c	c	b
4	e	d	d	d	c

Output:

4 1 1 4

Muddiest Points

- **Q: Not exactly clear on move to front encoding**

Input:

e a e d e e

0	a	e	a	e	d	e
1	b	a	e	a	e	d
2	c	b	b	b	a	a
3	d	c	c	c	b	b
4	e	d	d	d	c	c

Output:

4 1 1 4 1

Muddiest Points

- **Q: Not exactly clear on move to front encoding**

Input:

e a e d e e

0	a	e	a	e	d	e	e
1	b	a	e	a	e	d	d
2	c	b	b	b	a	a	a
3	d	c	c	c	b	b	b
4	e	d	d	d	c	c	c

Output:

4 1 1 4 1 0

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
- **Decoding**

Input:

4 1 1 4 1 0

0

a

1

b

2

c

3

d

4

e

Output:

e

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
- Decoding

Input:

4 1 1 4 1 0

0	a	e
1	b	a
2	c	b
3	d	c
4	e	d

Output:

e

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
- **Decoding**

Input:

4 1 1 4 1 0

0	a	e	a
1	b	a	e
2	c	b	b
3	d	c	c
4	e	d	d

Output:

e a

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
- Decoding

Input:

4 1 1 4 1 0

0	a	e	a	e
1	b	a	e	a
2	c	b	b	b
3	d	c	c	c
4	e	d	d	d

Output:

e a e 4 1 0

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
- Decoding

Input:

4 1 1 4 1 0

0	a	e	a	e	d
1	b	a	e	a	e
2	c	b	b	b	a
3	d	c	c	c	b
4	e	d	d	d	c

Output:

e a e d

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
- Decoding

Input:

4 1 1 4 1 0

0	a	e	a	e	d	e
1	b	a	e	a	e	d
2	c	b	b	b	a	a
3	d	c	c	c	b	b
4	e	d	d	d	c	c

Output:

e a e d e

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
- Decoding

Input:

4 1 1 4 1 0

0	a	e	a	e	d	e	e
1	b	a	e	a	e	d	d
2	c	b	b	b	a	a	a
3	d	c	c	c	b	b	b
4	e	d	d	d	c	c	c

Output:

e a e d e e

Muddiest Points

- **Q: LZW doesn't seem like prefix-free compression. Do we use delimiters between the codes or is there some way to tell the numbers apart. Even just 255 contains multiple interpretations of ASCII codes (2, 55 or 25, 5)**
- Great Question!
- Each integer is the same number of bits (e.g., 12 bits)
- So, the expansion program reads 12 bits at a time
- No need for delimiters nor prefix-free encoding of the integers

Muddiest Points

- **Q: calculating the bits for the lzw compression (last question on the tophat), we didn't go over this during the example in class**
- The second column represents the output of LZW compression, i.e., the compressed file
- Each integer is 12-bit codeword (per the question)
- Total compressed file size = # codewords * 12

Muddiest Points

- **Q: Corner case of lzw expansion**
- The tricky (corner) case happens when the longest match in compressions happens to be the string that was just added in the previous step
- Expansion sees that as a codeword that is not (yet) in its codebook
- Remember that expansion builds the same codebook as compression but is one step behind
- Handling the tricky case:
 - output: previous output + first character of previous output
 - add the same string to the codebook

LZW corner case example

- Compress, using 12 bit codewords: AAAAAA

Cur	Output	Add
A	65	AA:256
AA	256	AAA:257
AAA	257	--

- Expansion:

Cur	Output	Add
65	A	--
256	AA	256:AA
257	AAA	257:AAA

Muddiest Points

- **Q: How can the uncompressed file have more entropy than compressed if the entropy is the average number of bits to represent a word?**
- In lossless compression,
 - entropy of compressed file \geq entropy of uncompressed file
- Since compressed file has fewer characters than uncompressed
 - entropy/char of compressed file is $>$ entropy/char of uncompressed file

Muddiest Points

- **Q: I was wondering why LZW choose 12bits instead of any other number**
- 12 bits was used just as an example
- Actual implementation use an adaptive codeword size
 - start with 9 bits
 - when codebook full, change to 10 bits
 - when codebook full, change to 11 bits
 - ...
 - when codebook full and codeword is 16 bits
 - either stop adding to codebook or reset it
 - depends on the compression ratio

Muddiest Points

- **Q: please make assignments easier**
- Yep!

Muddiest Points

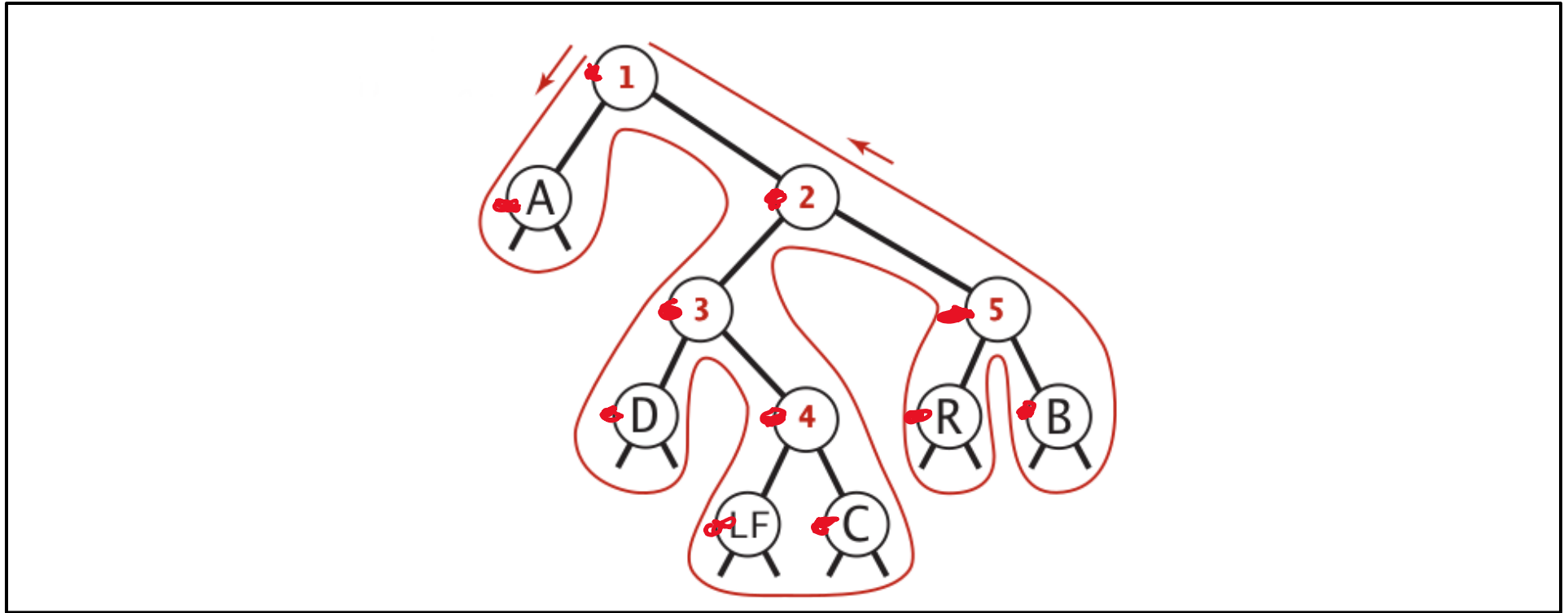
- **Q: Is there a best way to handle ties when doing Huffman compression or is it a purely arbitrary choice?**
- **Q: what are the rules for drawing out the tree in the huffman approach?**
- Ties are arbitrarily handled
- The compressed file size is the same no matter how ties are handled

Muddiest Points

- **Q: the bits needed for the compression post huffman encoding**
- Huffman compression may store the trie in the compressed file
- The trie is encoded using preorder traversal of the nodes
 - encoding internal nodes with 0
 - leaf nodes with 1 followed by the ASCII code of the character inside the leaf

Representing tries as bitstrings

Preorder traversal

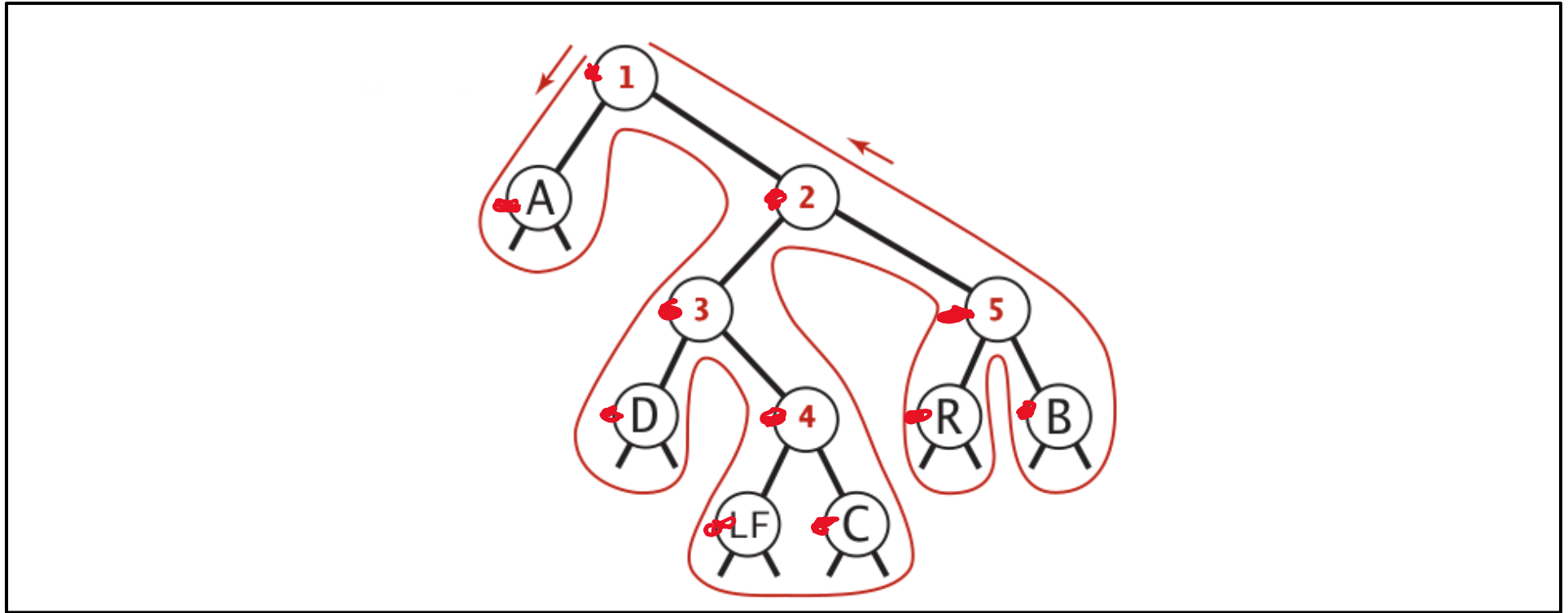


internal node \rightarrow 0

leaf node \rightarrow 1 followed by ASCII code of char inside

Representing tries as bitstrings

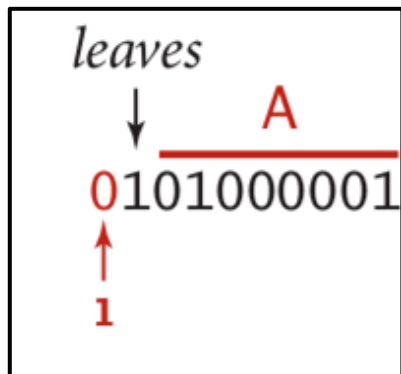
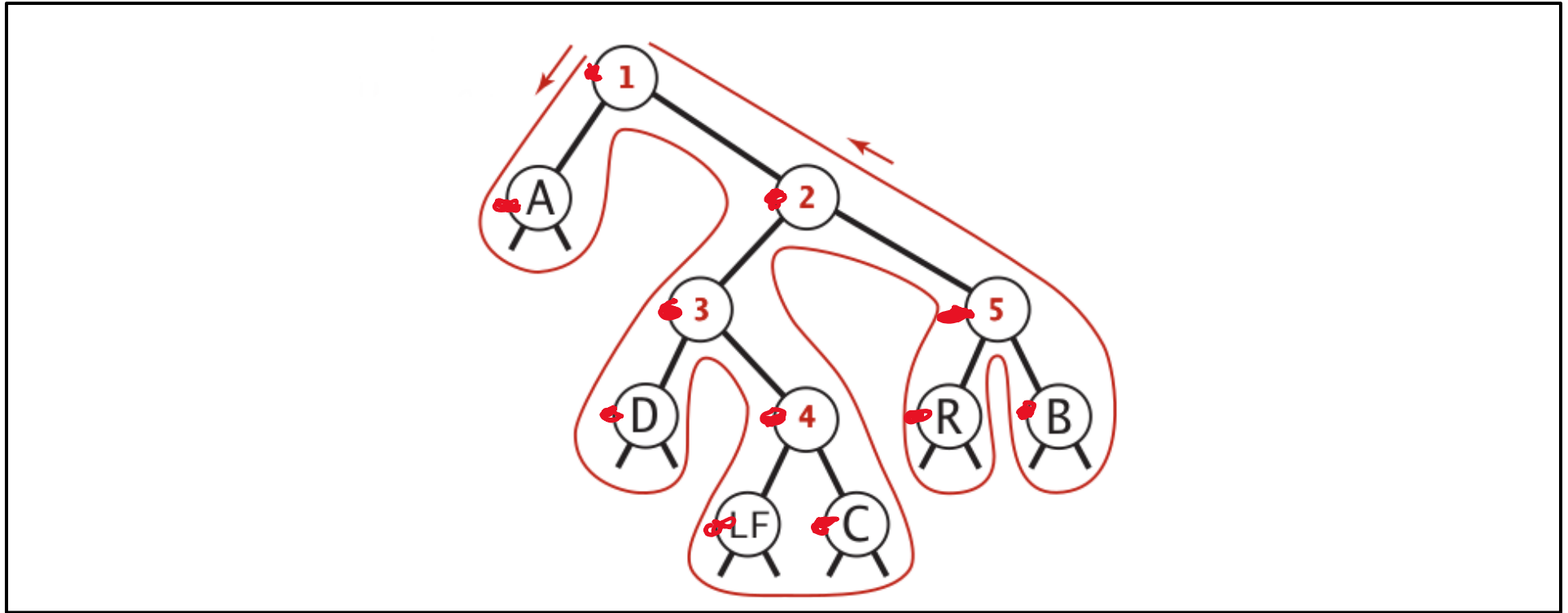
Preorder traversal



lea
0
↑
1

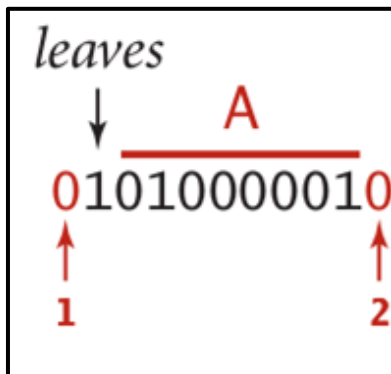
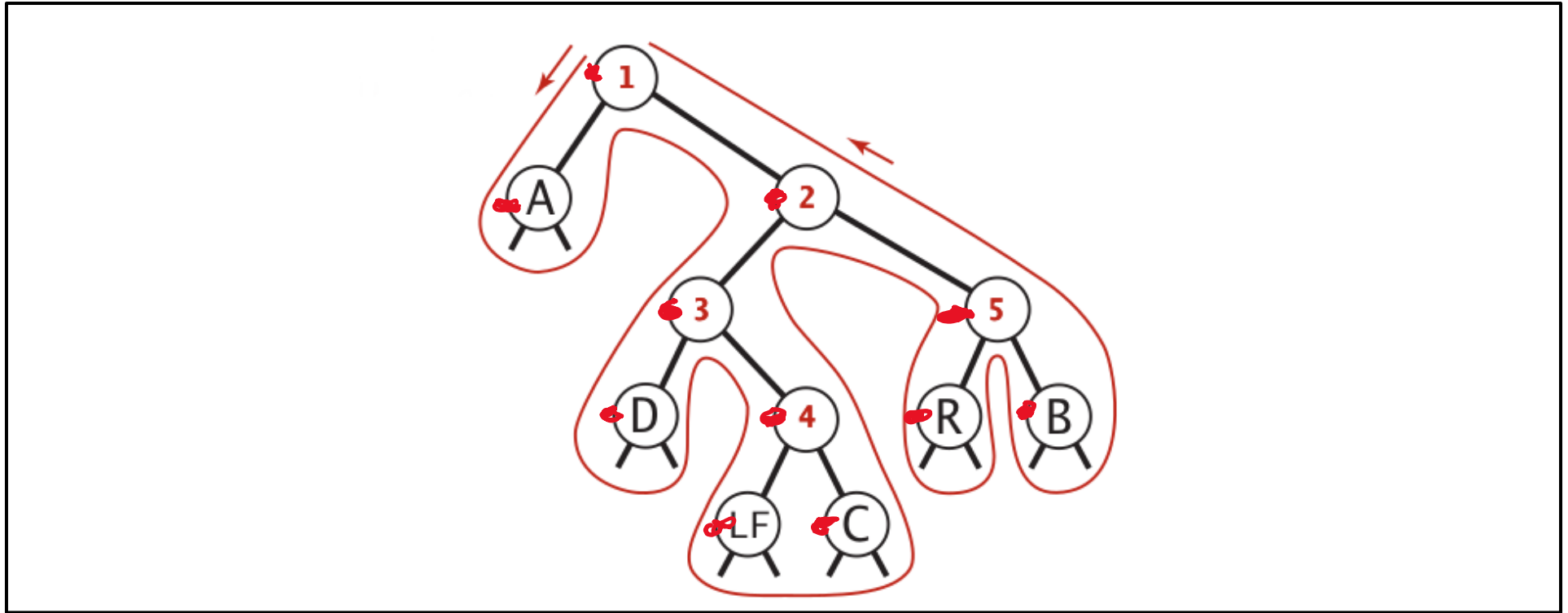
Representing tries as bitstrings

Preorder traversal



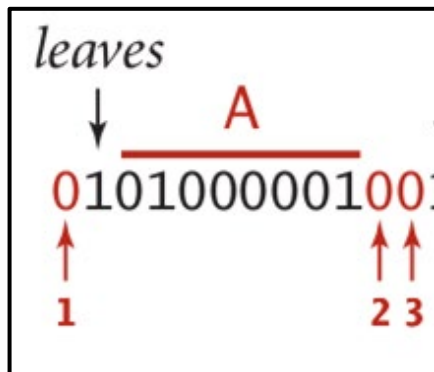
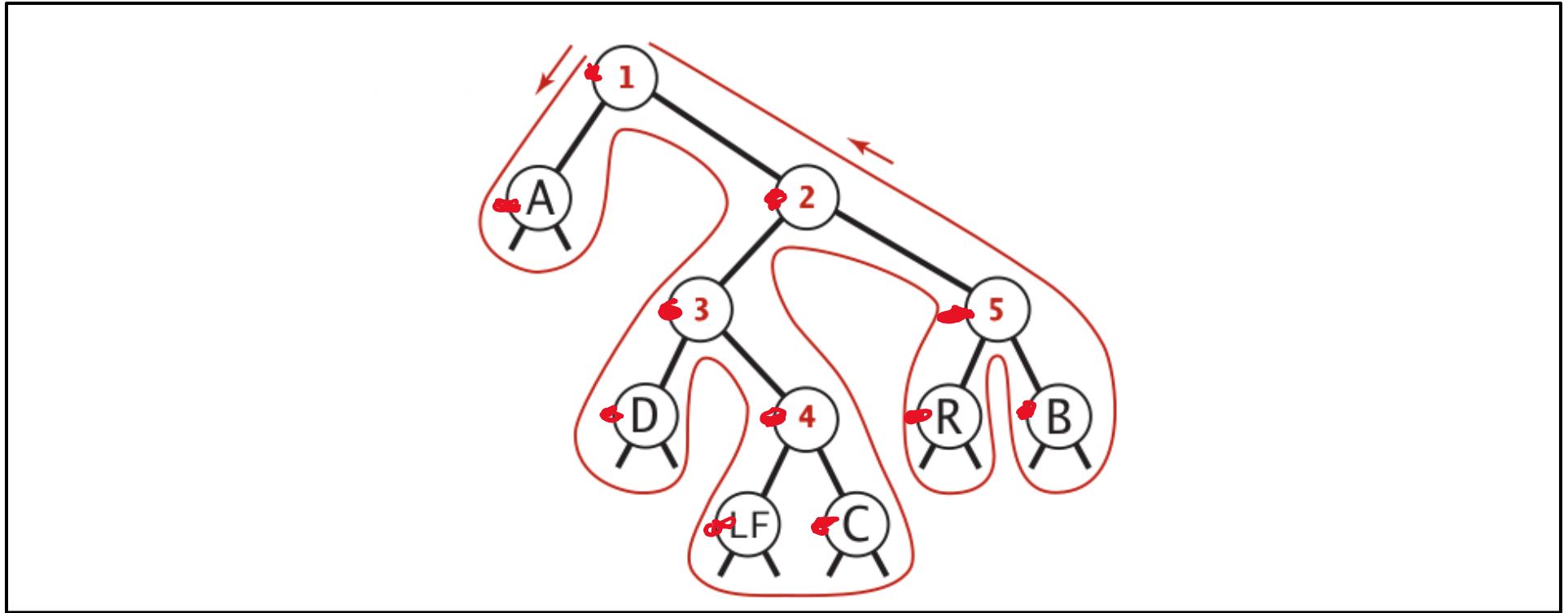
Representing tries as bitstrings

Preorder traversal



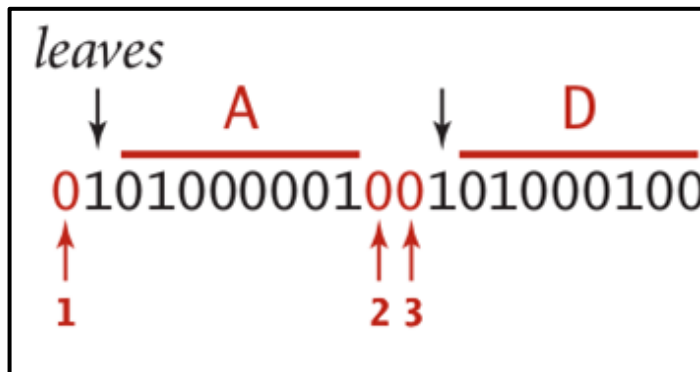
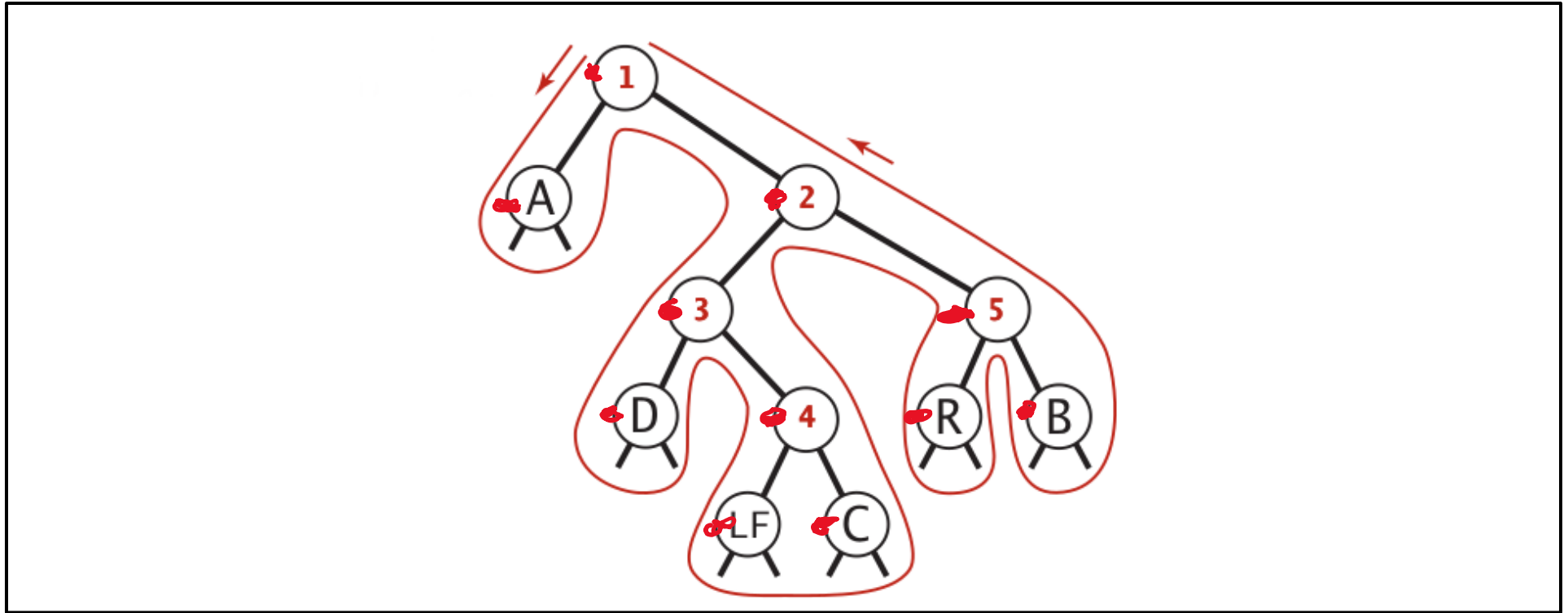
Representing tries as bitstrings

Preorder traversal



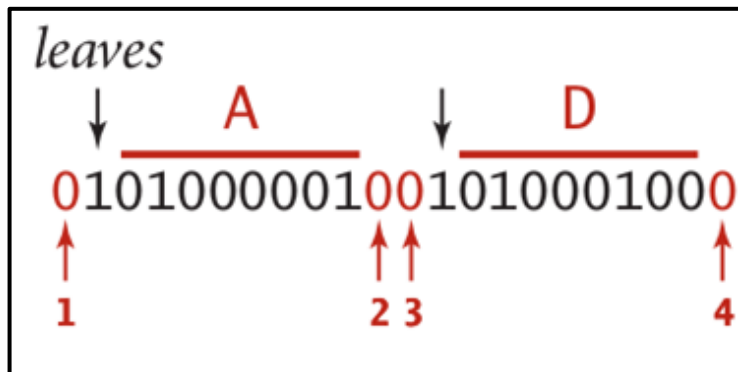
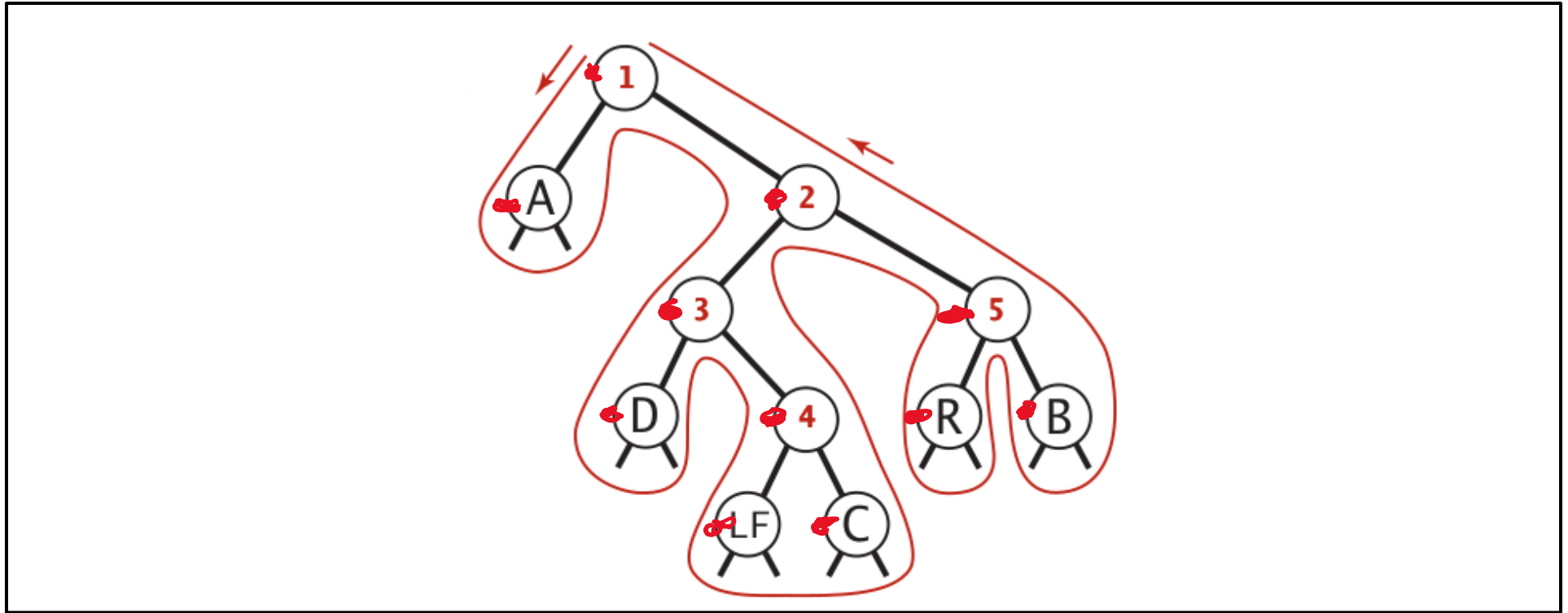
Representing tries as bitstrings

Preorder traversal



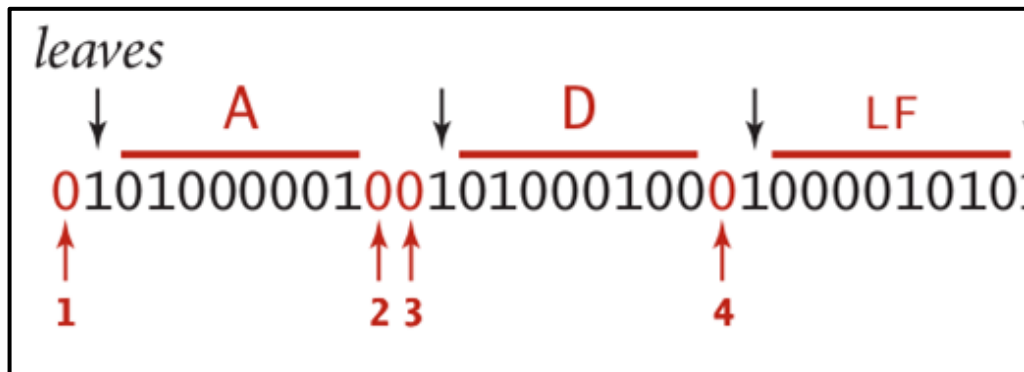
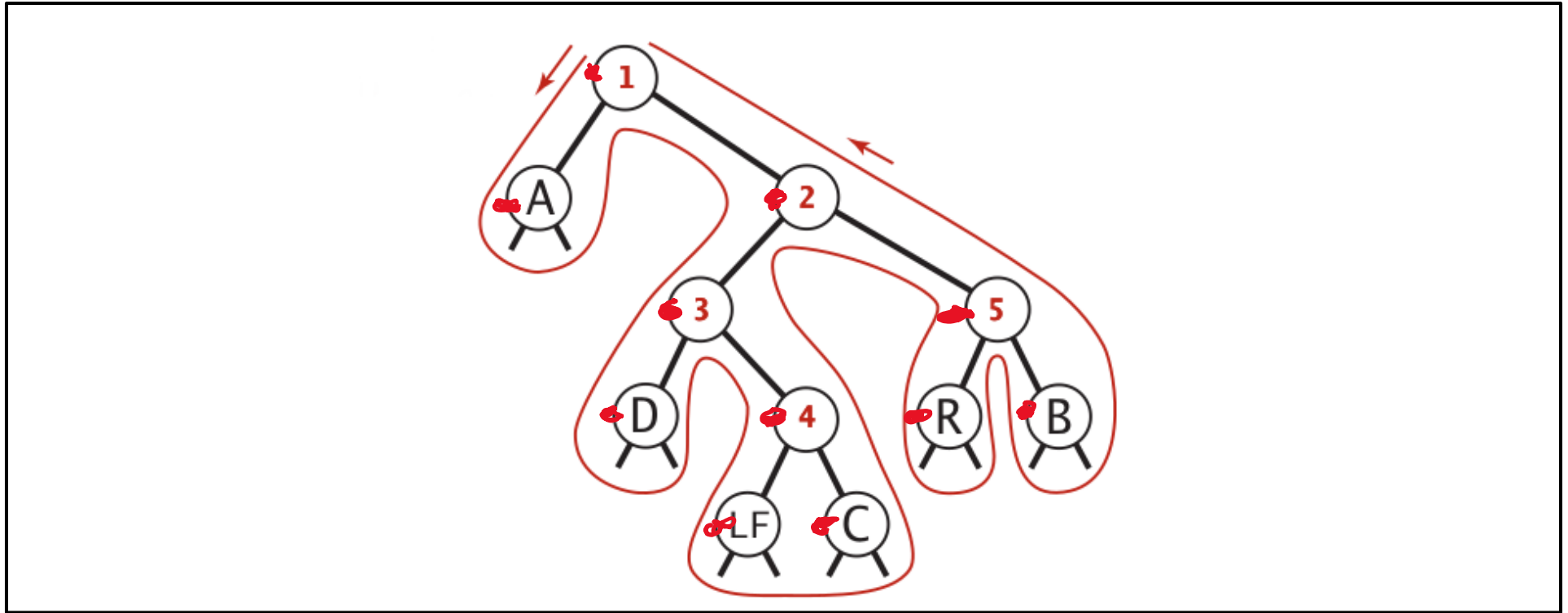
Representing tries as bitstrings

Preorder traversal



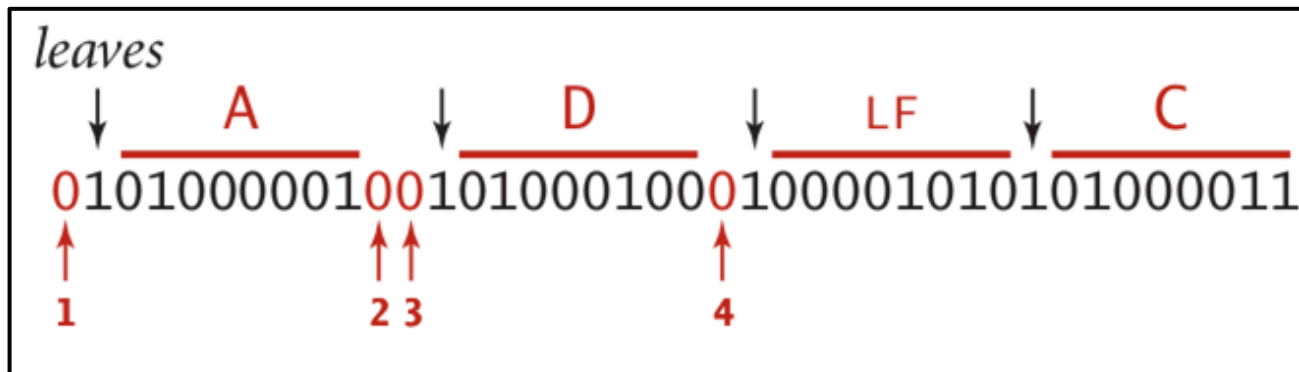
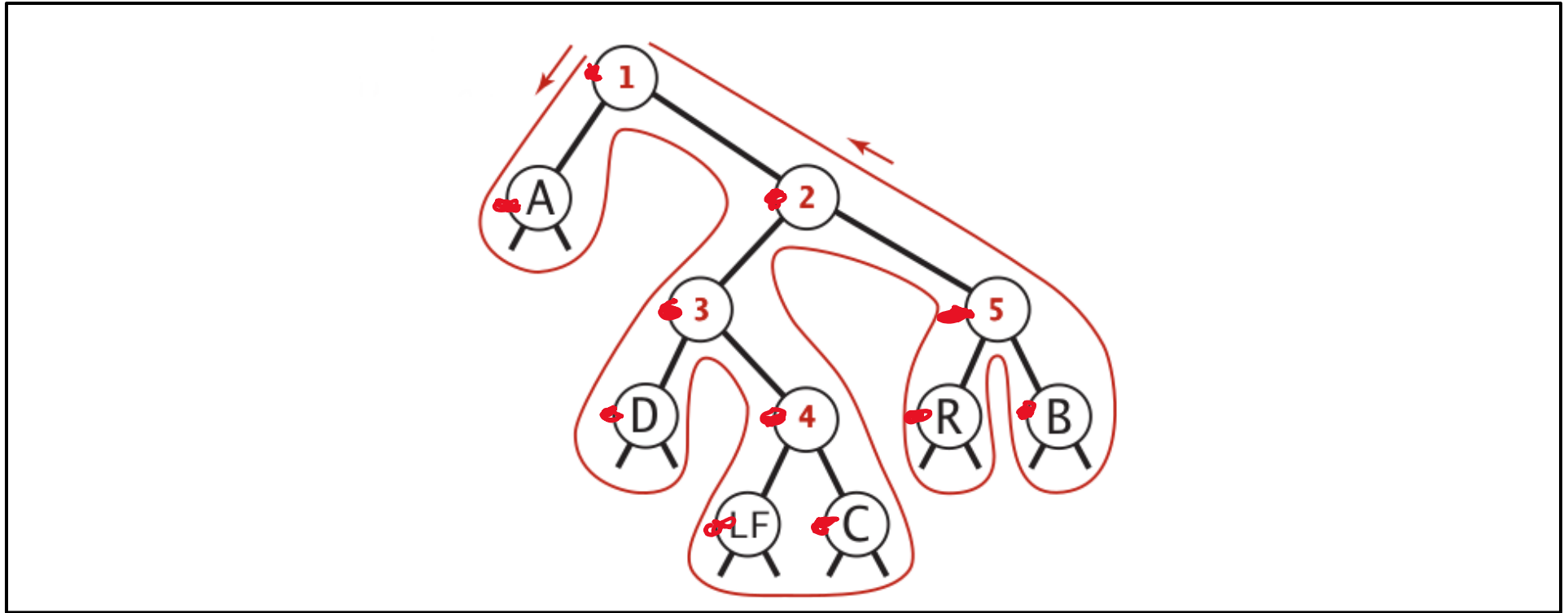
Representing tries as bitstrings

Preorder traversal



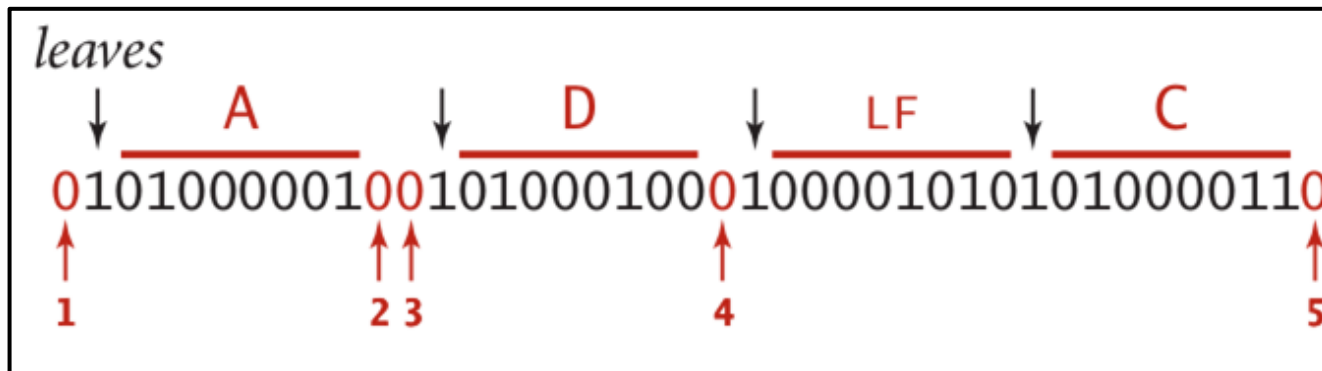
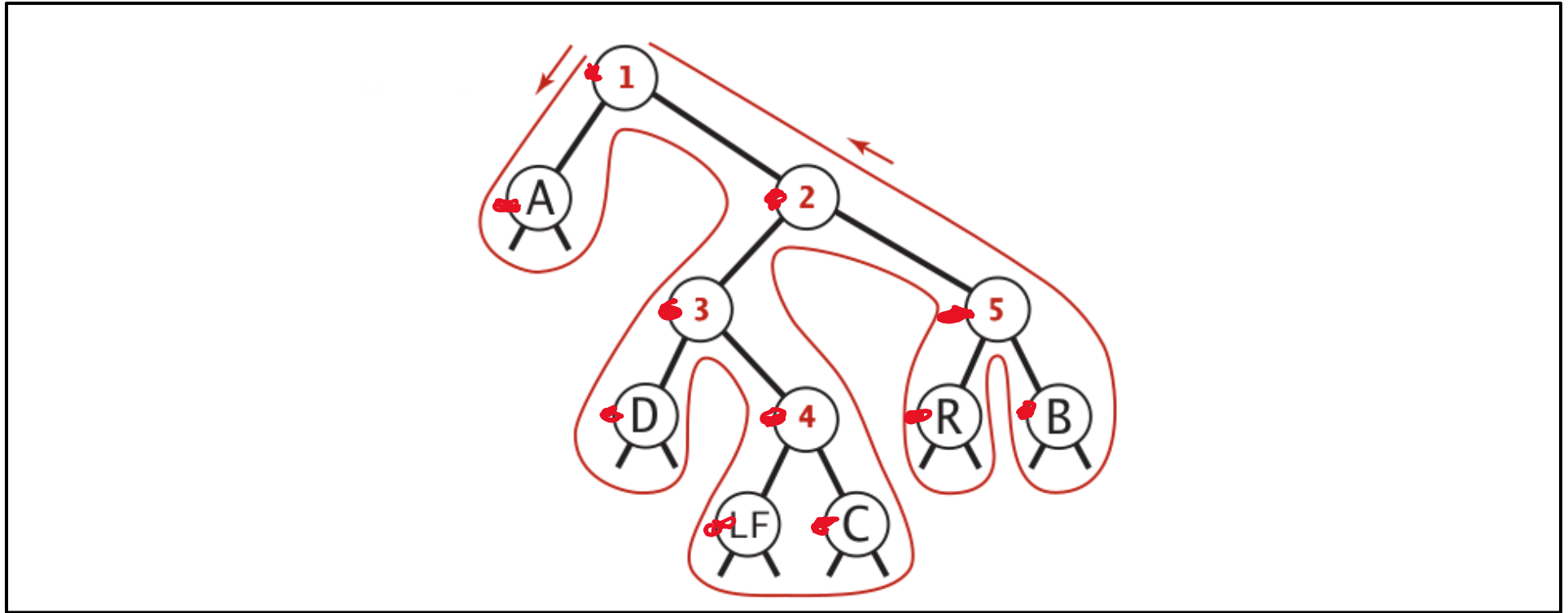
Representing tries as bitstrings

Preorder traversal



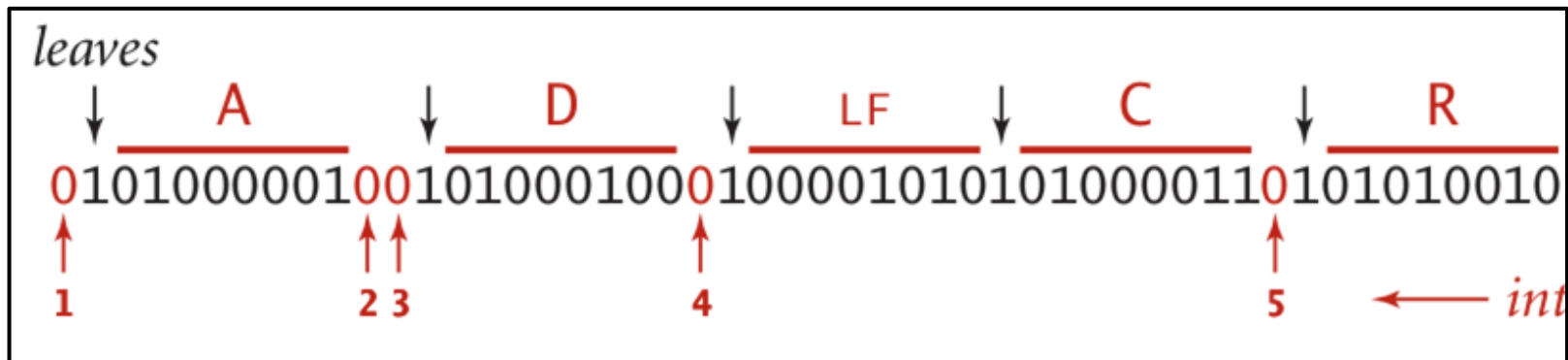
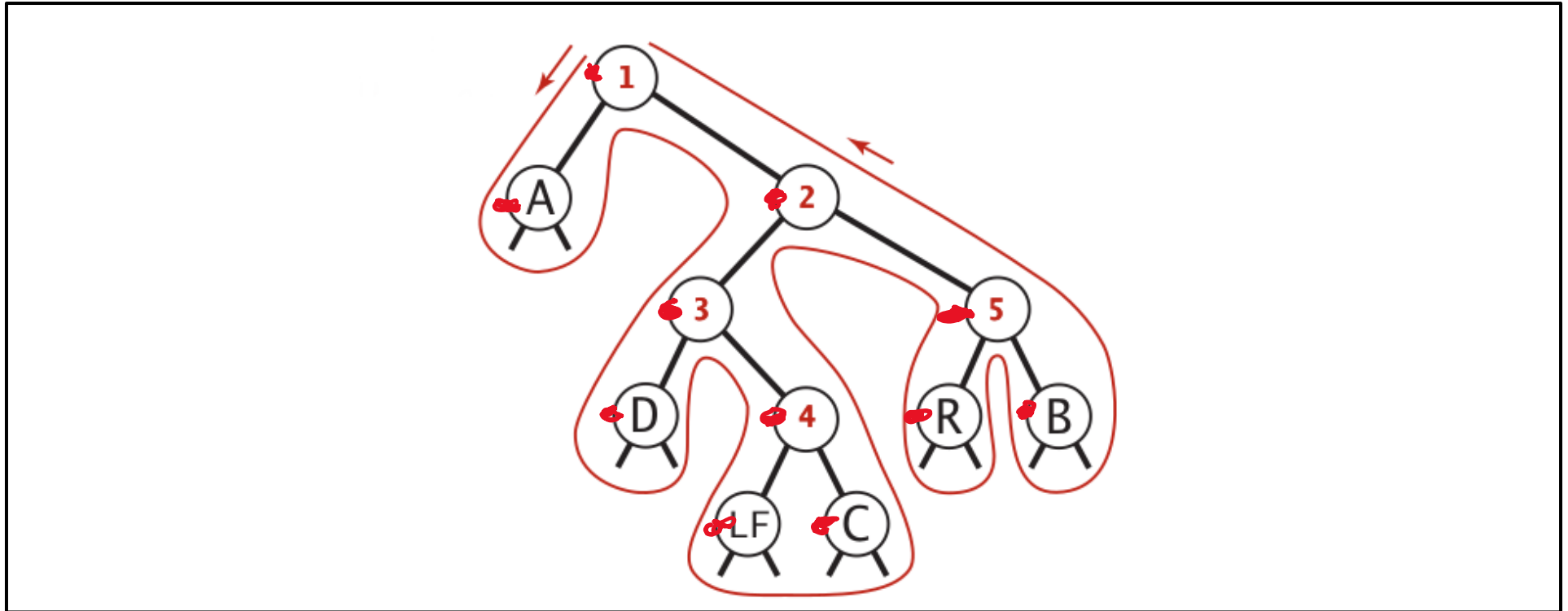
Representing tries as bitstrings

Preorder traversal



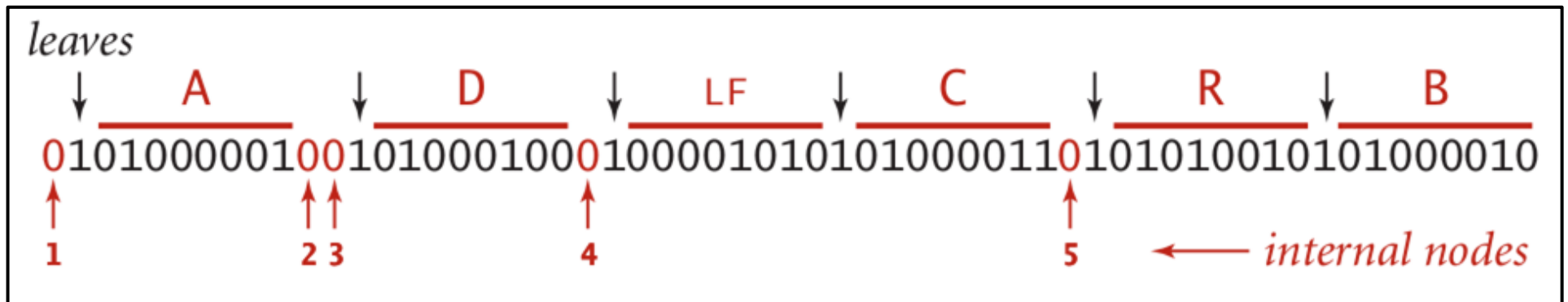
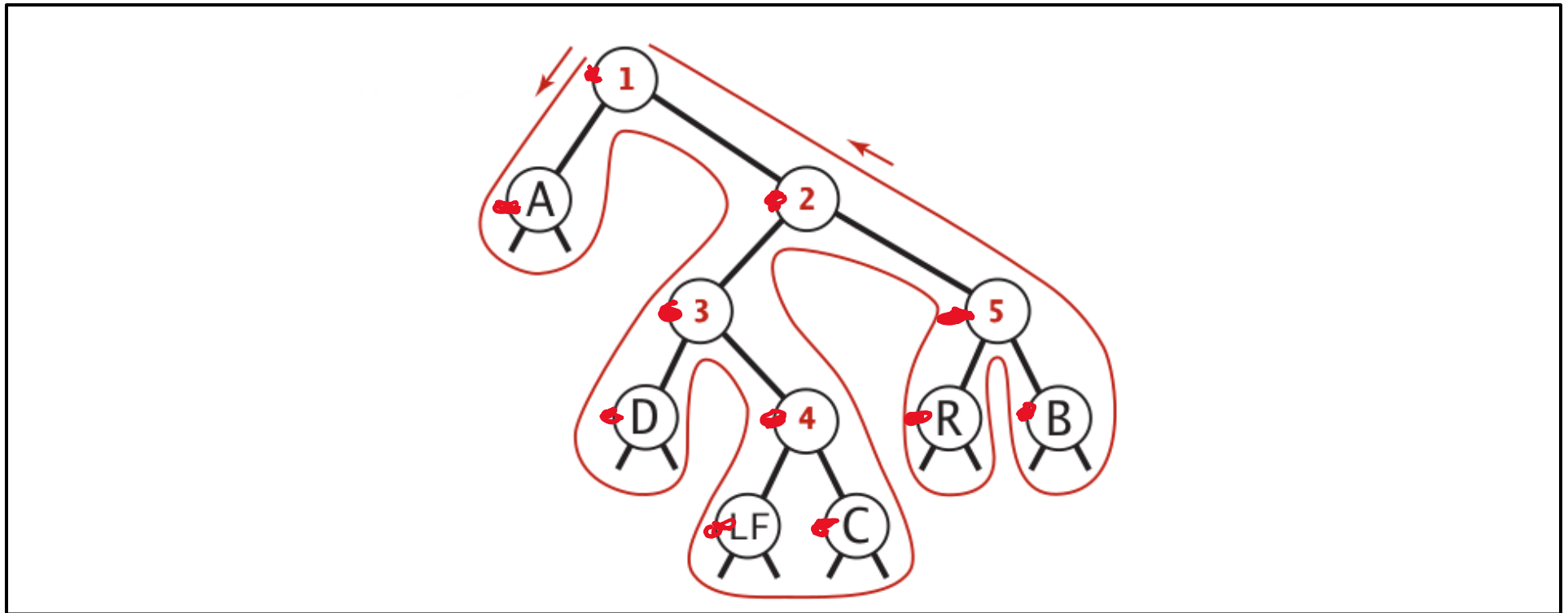
Representing tries as bitstrings

Preorder traversal



Representing tries as bitstrings

Preorder traversal



Muddiest Points

- **Q: After writing out the Trie as a bitstring, how can you tell where each character starts?**
- First, we are writing bits not characters!
- Each bit represents a node
 - except for leaves, their 1 bit is followed by 8 bits (ASCII code of char inside)
 - but that's also fixed length

Muddiest Points

- **Q: how compression and tries combine. Are you supposed to recreate the original trie while decompressing**
- The compressed file contains:
 - the trie representation in bits
 - the number of characters in the original file
 - the Huffman encoding of the file characters
- The original trie is reconstructed from the trie representation in the compressed file

Muddiest Points

- **Q: Questions are not getting responded to on Piazza in a timely manner. Students had asked several questions pertaining to items on Homework 4 last week but the questions were not addressed until AFTER Homework 4 was already graded. The questions I got marked wrong this week were the exact questions I had asked for clarification on but didn't get a response to (until after the homework was graded and returned...by that point it was too late).**
- I will hold a Live QA Session on Piazza every Friday 4:30-5:30 pm

Muddiest Points

- **Q: the overall concepts of code blocks/code words, like how do they fit into everything?**
- The input file is divided into code blocks
- Each code block is replaced by a codeword
- For Huffman:
 - code blocks are single characters
 - codewords are variable-length bit strings
- For LZW:
 - code blocks are the longest-match strings (variable length)
 - codewords are fixed-length integers (e.g., 12 bits)
- For RLE:
 - code block are long strings with identical characters (variable length)
 - codewords are fixed-length integer followed by fixed-length ASCII of the character

Muddiest Points

- **Q: Just making sure I got the order of the compression framework: We go from file to code block to code word via compression then back to code block via expansion**
- Yep!

Muddiest Points

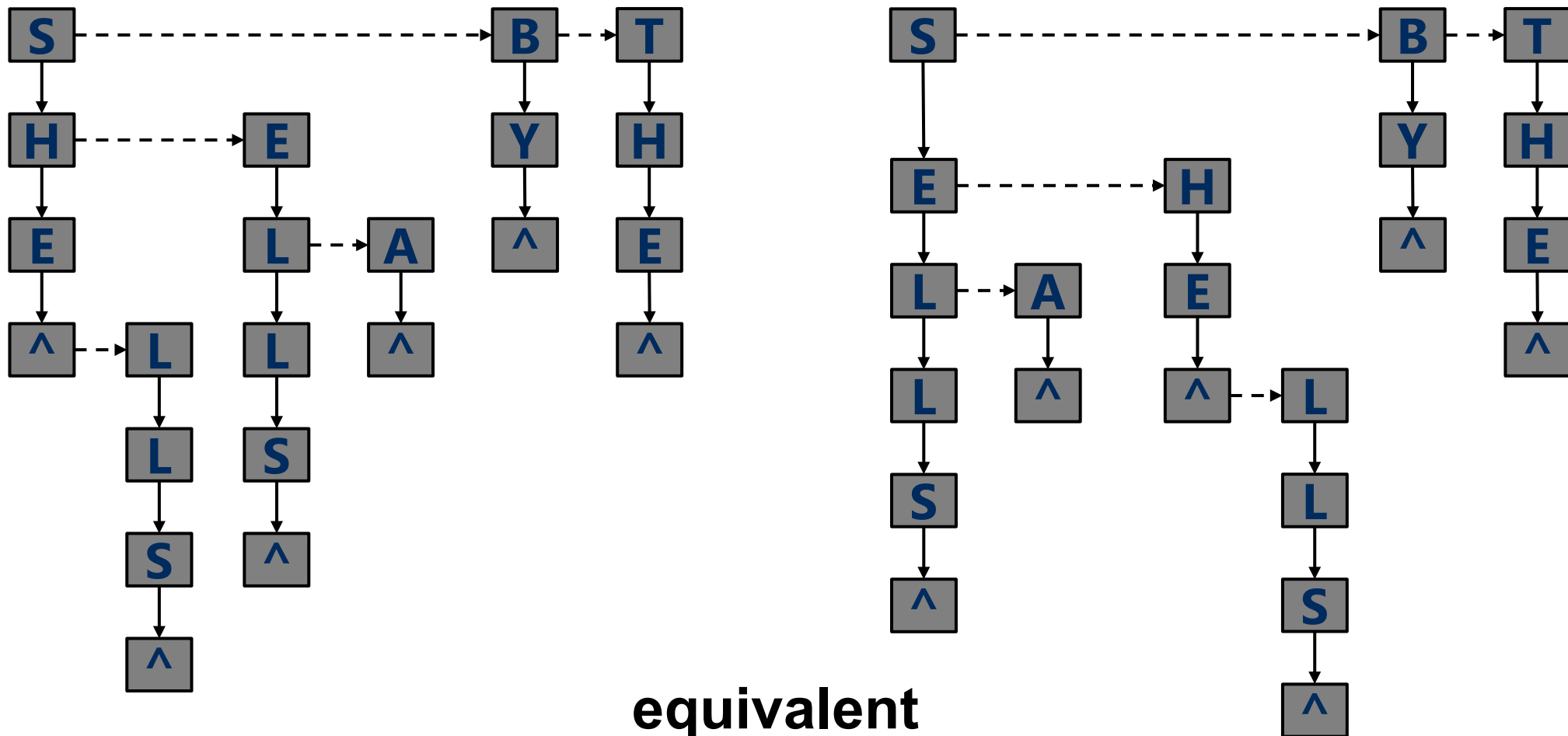
- **Q: Can we get partial credit on labs and projects?**
- We assign partial credit on projects only and only when autograder score $\leq 40\%$
 - partial credit has a ceiling of 60% of the autograder score
- Unless you think you lost points in the autograder because of an autograder error

Muddiest Points

- **Q:How are you able to create unique codes without leading to another prefix**
- Characters are leaves in the Huffman tree
- No two characters share the same path from root
- Codewords encode the root-to-leaf path
- No two codewords share a prefix

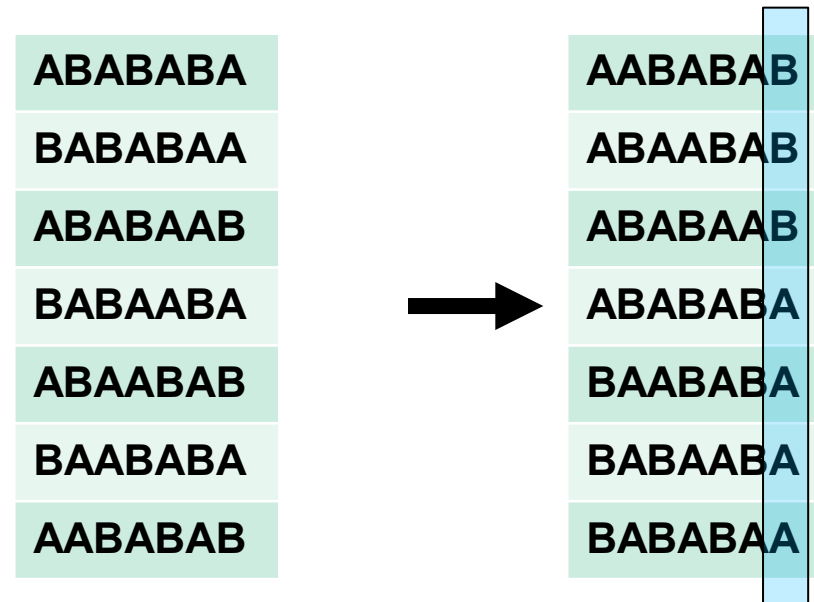
Muddiest Points

- **Q: In DLB tries, can you interchange a parent node's child with any one of the child's siblings? I think you can but you would have to change the sibling links, is that correct?**
- Correct!



Burrows-Wheeler Transform Example

- ABABABA
- Step 1: Build an array of 7 strings, each a circular rotation of the original by one character
- ABABABA
- BABABAA
- ABABAAB
- BABAABA
- ABAABAB
- BAABABA
- AABABAB
- Sort the array alphabetically



Output of BWT:
BBBAAAA and 3