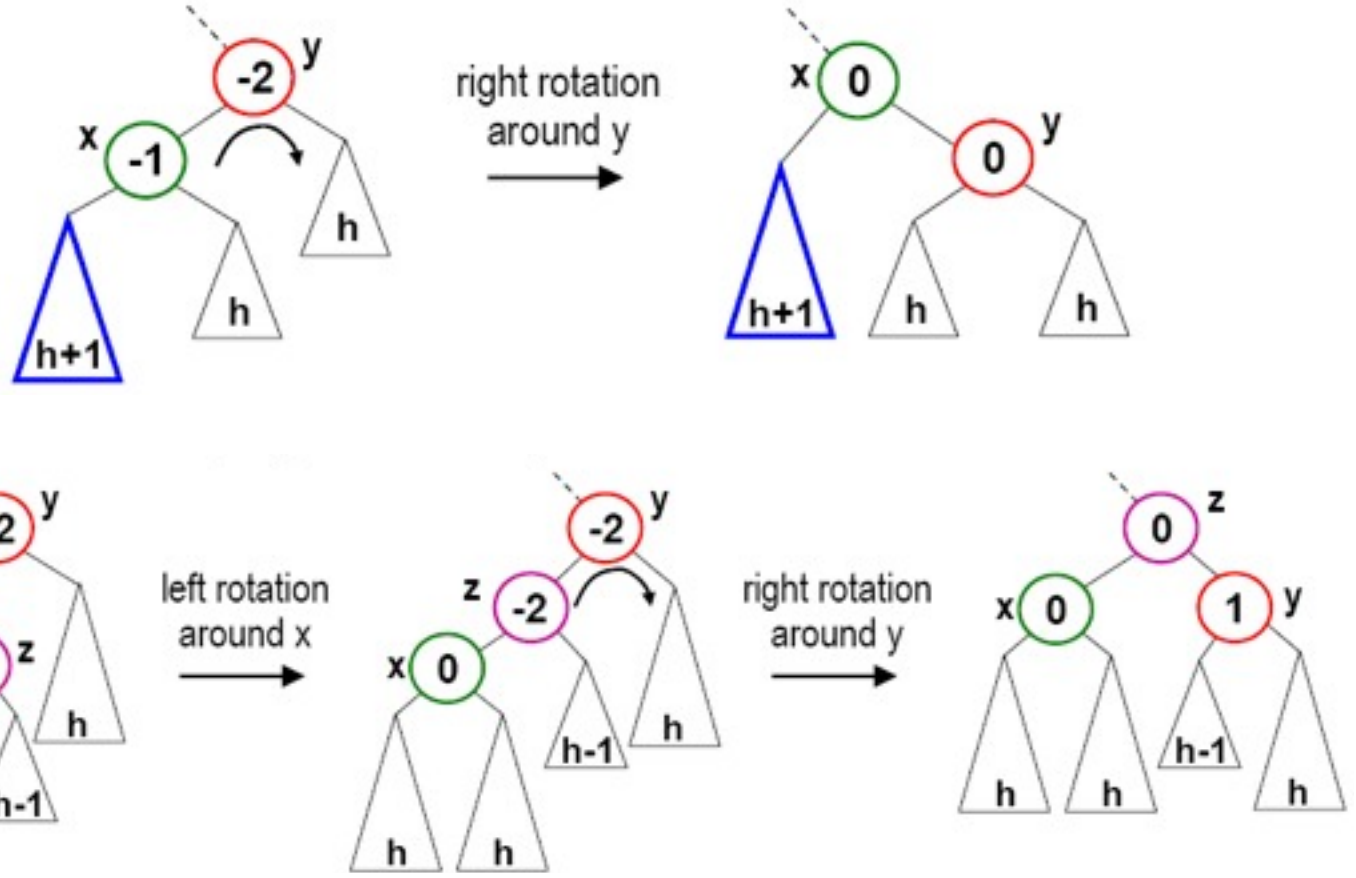
Other Balanced Trees, and an Unbalanced One (Huffman Encoding)

EECS 233

Previous Lecture

- AVL trees (self-balancing binary tree)
- Rotation operations: left rotation, right rotation, double rotation



Previous Lecture

- Insertion may cause change of balance value
- Insertion algorithm
  - Insert a node, which may cause a change to balance values
  - For each ancestor (from the leaf to the root), if the balance value
    - ✓ changes from +/-1 to 0, DONE
    - ✓ changes from 0 to +/- 1, go to the next ancestor
    - ✓ changes from +/-1 to +/- 2, rebalance this tree, DONE
  - How to balance the tree? If insertion occurred in the
    - ✓ left-left subtree, do right rotation
    - ✓ right-right subtree, do left rotation
    - ✓ left-right subtree, do left-right double rotation
    - ✓ right-left subtree, do right-left double rotation

Deletion from AVL Trees

- Deletion may also cause unbalanced subtrees; if a deletion causes a change to the height, the ancestors need to be checked or balanced.

- Deletion algorithm
  - remove the node (leaf or non-leaf, remember how it works?)
  - Ultimately causes removal of a node X with 0 or 1 child
  - May cause a change of the balance values for X's parent

  - For each ancestor (from X's parent to the root), if the balance value
    - ✓ changes from +/-1 to 0, what to do?
    - ✓ changes from 0 to +/- 1, what to do?
    - ✓ changes from +/-1 to +/- 2, what to do?

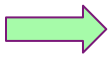
  - To perform the balancing, use the same rotations as in insertion

Efficiency of AVL Trees

- An AVL tree containing n items has a height that is $O(\log_2 n)$.
- Search and insertion are both $O(\log_2 n)$.
  - Search travels at most one path down the tree
  - An insertion goes down one path to the insertion point, and then goes back up adjusting balances/performing rotations
    - ✓ in the worst case, both the path down and the path back up consider $O(\log_2 n)$ nodes
    - ✓ each rotation requires a constant number of operations, and we perform at most two of them for a given insertion
- Deletion begins with the same procedure used in binary search trees ($O(\log_2 n)$ complexity). It can also require rotations to rebalance the tree, also at the complexity of $O(\log_2 n)$.

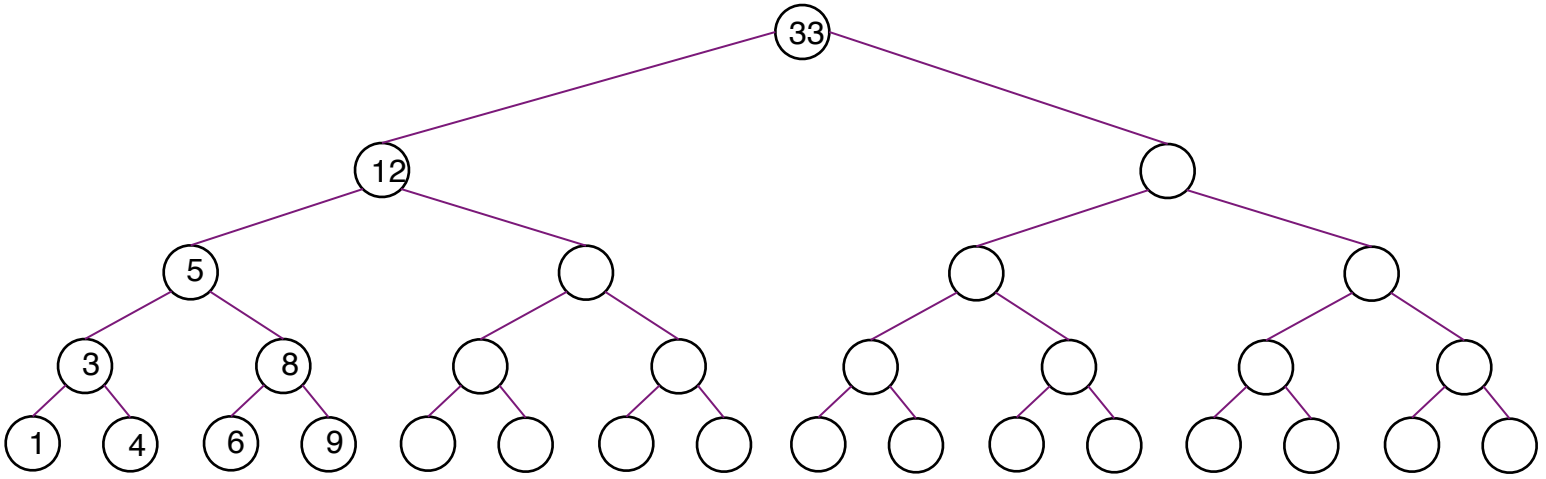
Other Trees

- 2-3 trees
- Splay trees
- Red-black trees
- B-trees



Aren't AVL Trees Perfect?

- AVL tree - binary balanced tree
  - Searching an item requires traversing from root to (at most) leaf
  - logN descents (random accesses)
  - Visiting each node - one comparison.
- On disk:
  - Random access is expensive
  - Disk read is in blocks (e.g. 4K bytes at once)
  - Disk-based AVL trees waste much of a block for an internal node read
    - ✓ The node is smaller than the block
    - ✓ Payload of the node in not used



B-tree motivations

- not all operations take equal time
  - CPU: 3 billion operations per second
  - Hard Disk access: ~9 ms to read a block
    ⇒ ~111 accesses per second
  - (newer SDDs: 0 ms seek time, latency .09 ms
    ⇒ have to design new search trees to match new characteristics)
- Each disk access = 27 million CPU operations

Search trees in the real world

- Typical large database:
  - 10 million records
  - each key: 32 bytes (e.g. name)
  - each record: 256 bytes
  - ~ 3 TB of data
- How much time to we have to access?
  - We're one of many users on system, say 30
  - in 1 sec we get 100 million CPU operations and about 4 disk accesses.

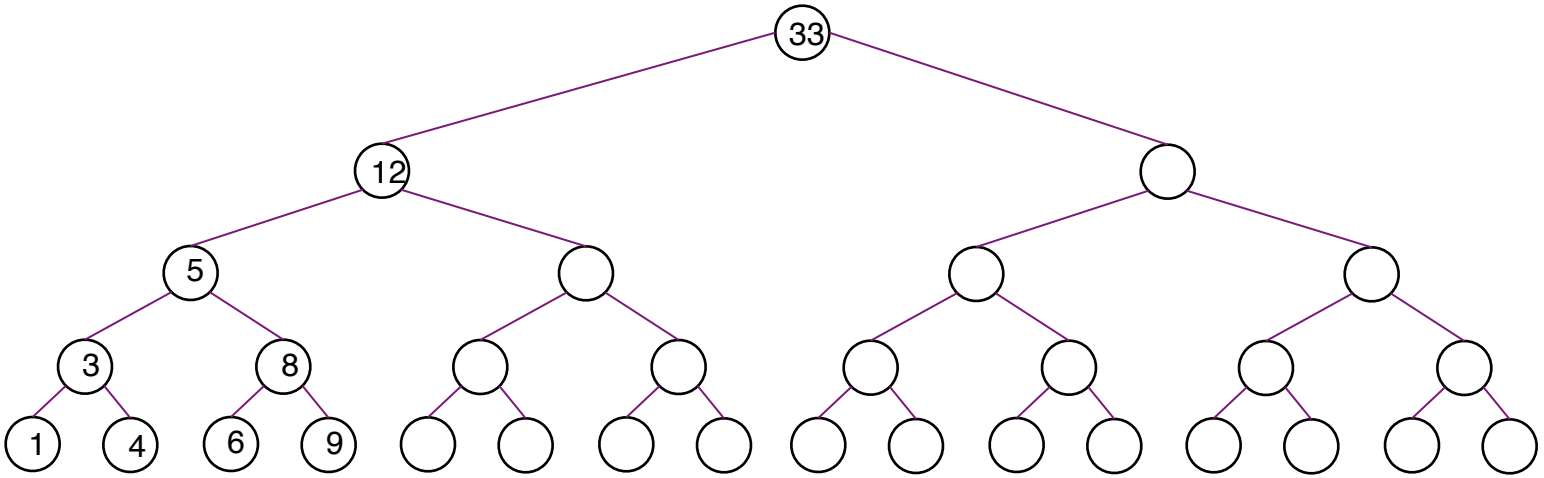
What can we do with what we have?

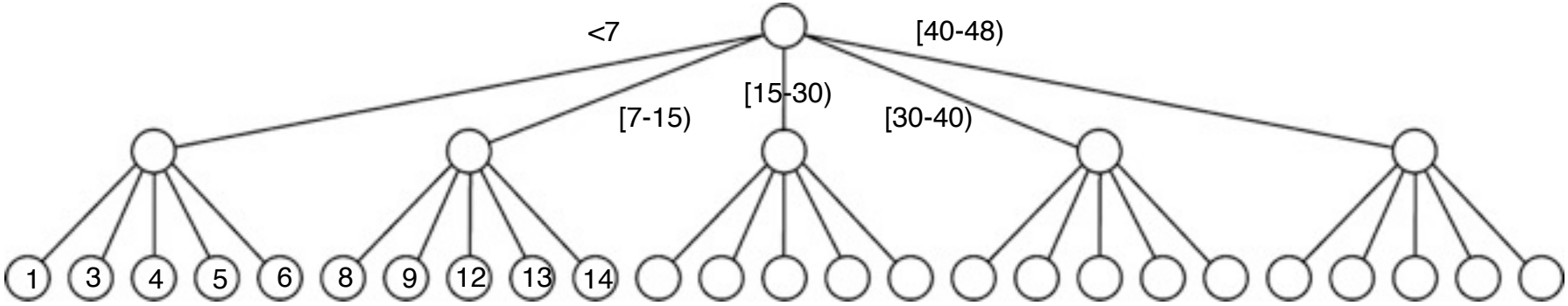
- 100 million operations and 4 disk accesses
- unbalanced binary tree:
  - 10 million disk accesses * 4/sec = a *lot* of time (1.24 years!)
- perfectly balanced binary tree
  - On average, need 1.38 log N disk accesses
  - $\log_2$ 10 million ≈ 24
  - So about 32 disk access total = 8 seconds
- typical binary tree: a few nodes are several times deeper
  - Could require several times 8 seconds
- AVL tree: typical case very close to log N
  - Still 6 seconds for each search

What can we do?

- Design better search tree:
  - requires no more than 3 or 4 disk access
  - don't care (much) about CPU operations
  - essentially free compared to disk accesses
- can make search and data structures more complex provided we minimize the disk accesses
- Conclusion:
  - can't have deep trees
  - need wide, shallow trees

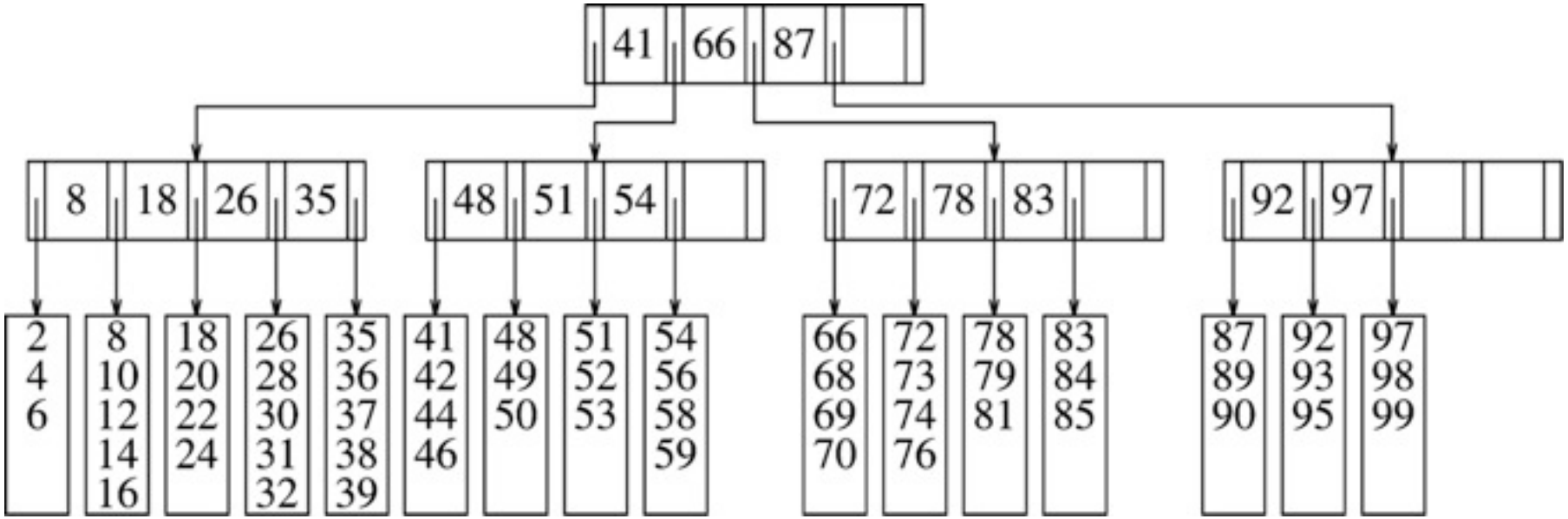
Designing for Disk Particulars





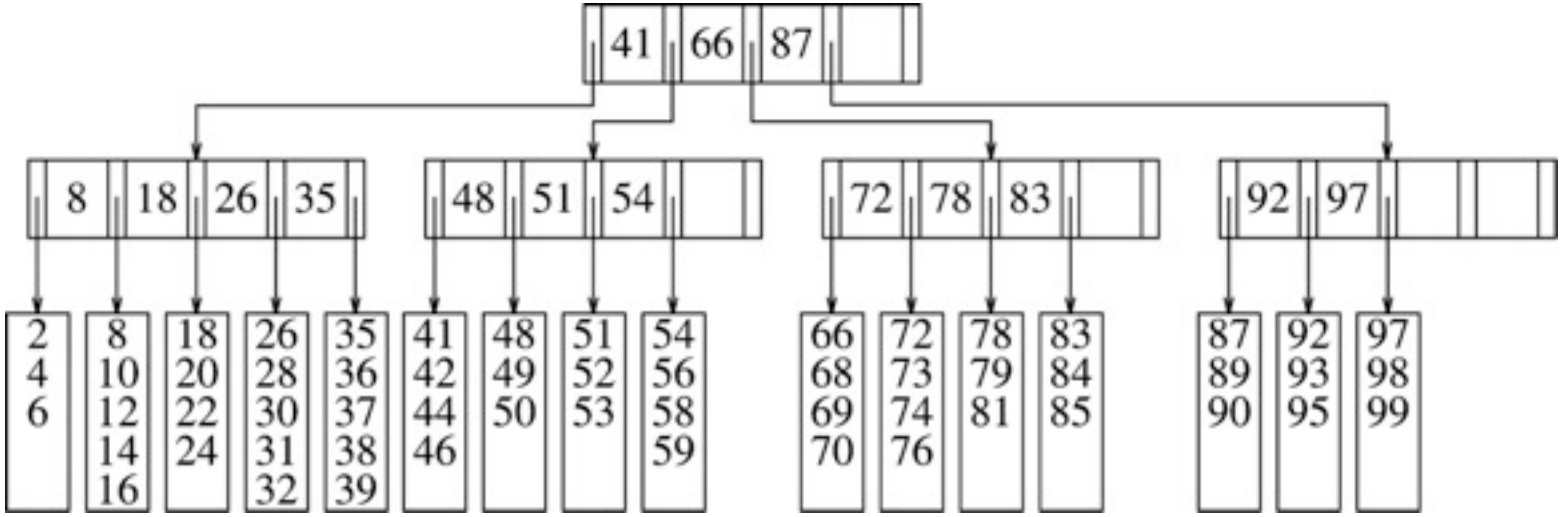
B-Tree Example

- Order: branching M=5, items in leaf L=5
- Example: search 74



B-Tree Example

- A B-tree of order *M* is a balanced tree in which
  - Each non-leaf node stores up to *M*-1 keys and *M* pointers; key i represents the smallest key in subtree i+1 (pointer i+1)
  - Data items are stored in leaf nodes (L=maximum number of items.)
  - Each node except for root is at least half full
    - ✓ Non-leaf node: at least M/2 pointers
    - ✓ Leaf node: at least L/2 entires
  - The root is either a leaf, or has between 2 and *M* children
- Determining M and L
  - (M-1) * sizeof(key) + M * sizeof(pointer) <= Block size (e.g. 4KB)
  - L * sizeof(item) <= Block size



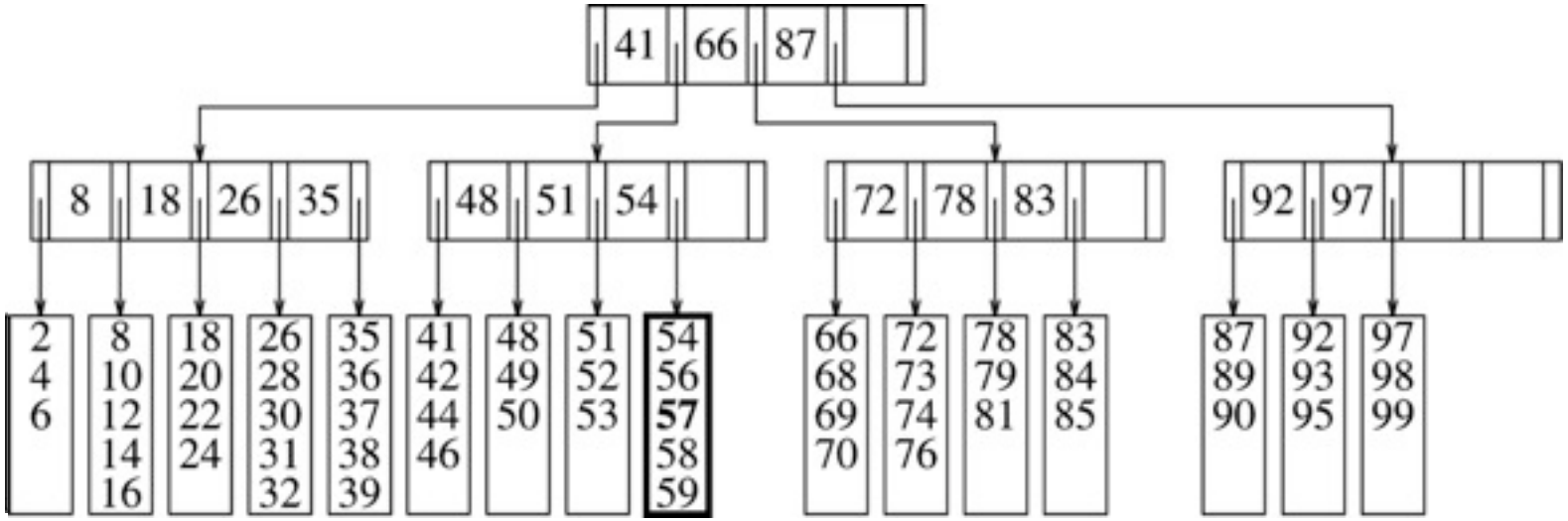
Determine B-tree parameters

- Example: 10 million records; 8k = 8,192 byte blocks
  - block size depends on application (e.g. names vs photos)
- Each node is a block
- B-tree of order M has
  - M-1 keys: each key is 32 bytes (e.g. a name) = 32*M - 32 bytes
  - M branches: each branch is another block (4 bytes) = 4*M bytes
- How large can M be?
  - $36*M - 32 < 8{,}192 \Rightarrow M = 228$
- Suppose each record is 256 bytes
  - can fit 8192/256 = 32 records in one block
- B-tree parameters
  - each branch node branches at least M/2 = 114 ways
  - between 16 and 32 records per block at leaf nodes

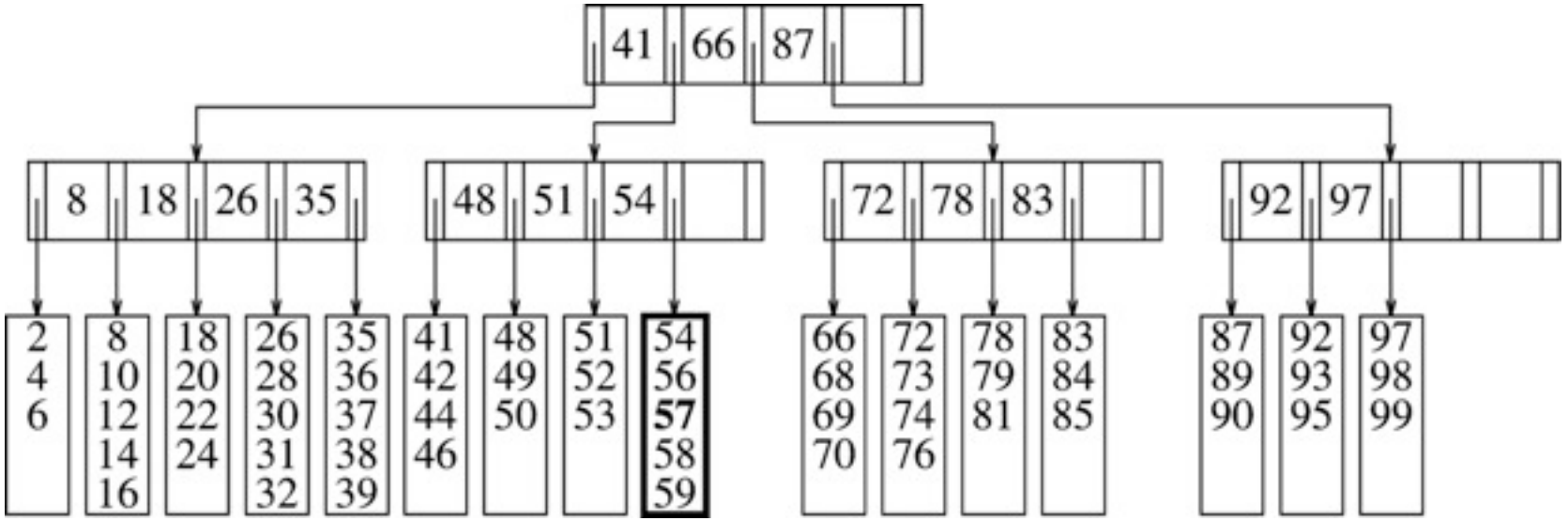
Insertion into B-Trees

- First search for the key until we reach a leaf
- Insert the key into the array of the leaf node
- … assuming there is room

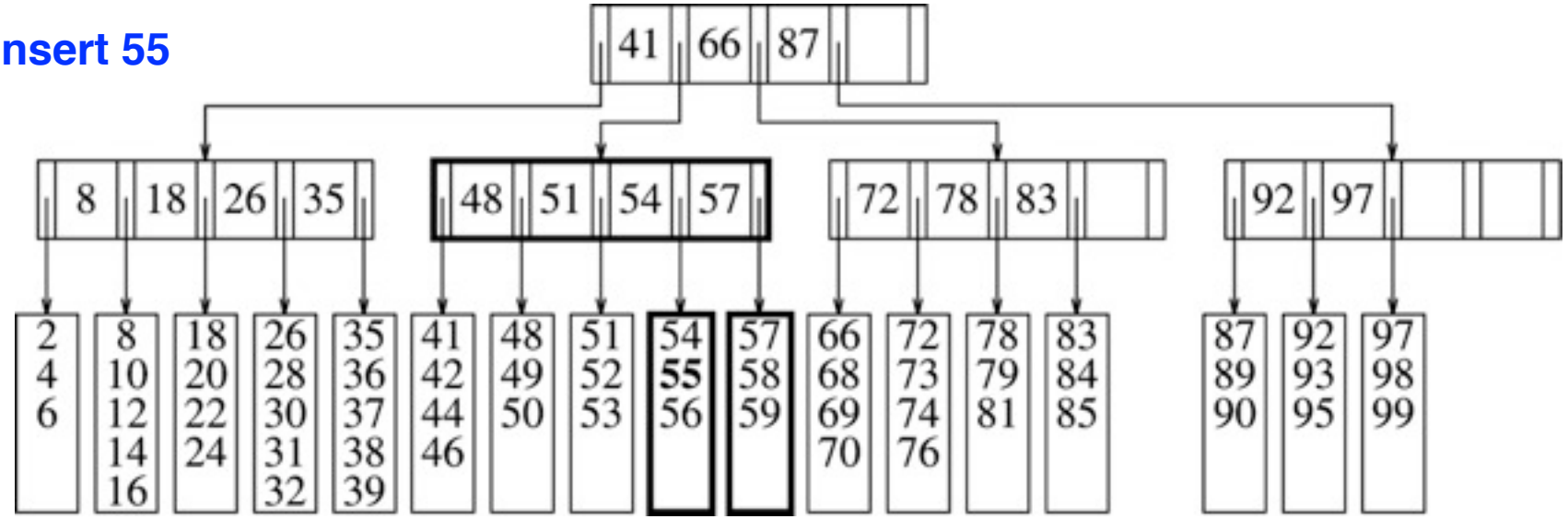
Insert 57



Leaf Split due to Insertion

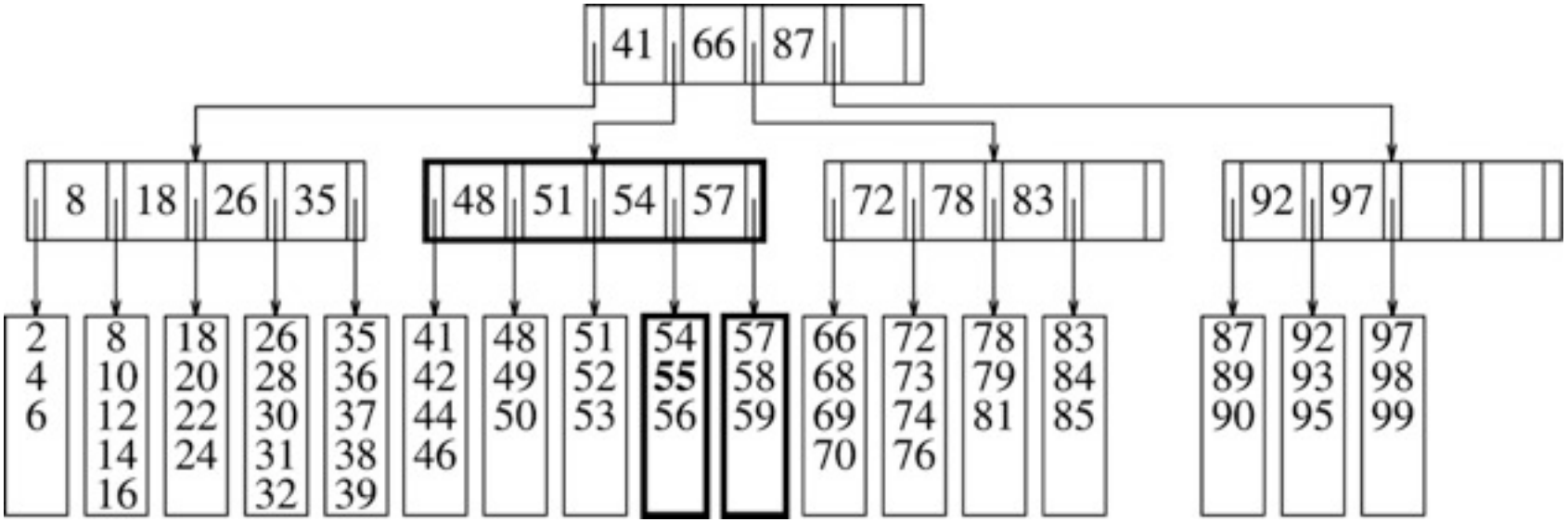


Insert 55

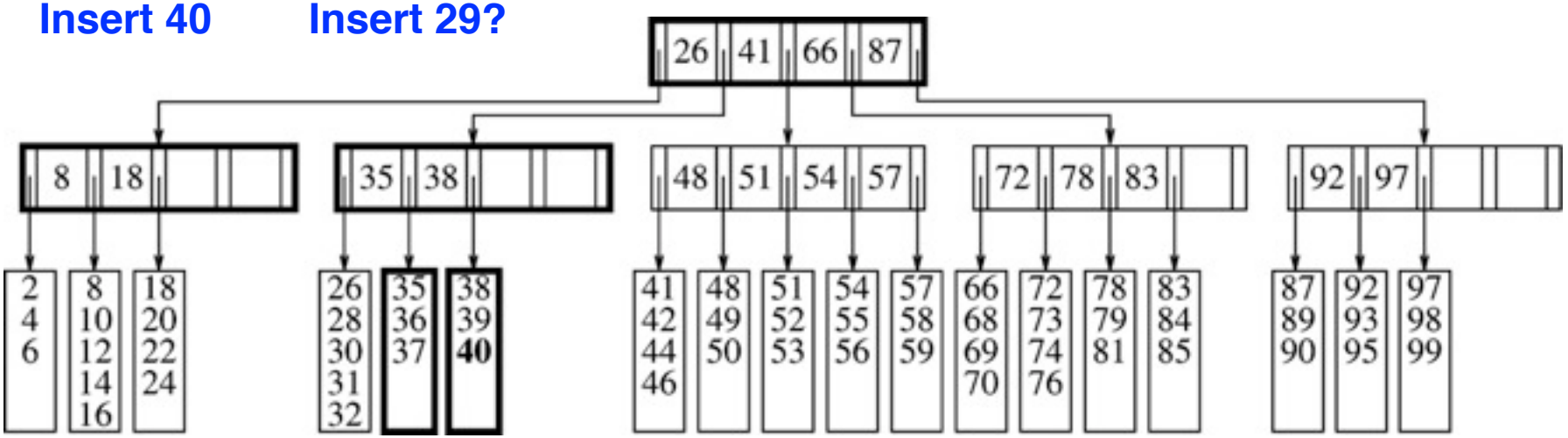


--

Splitting an Internal Node



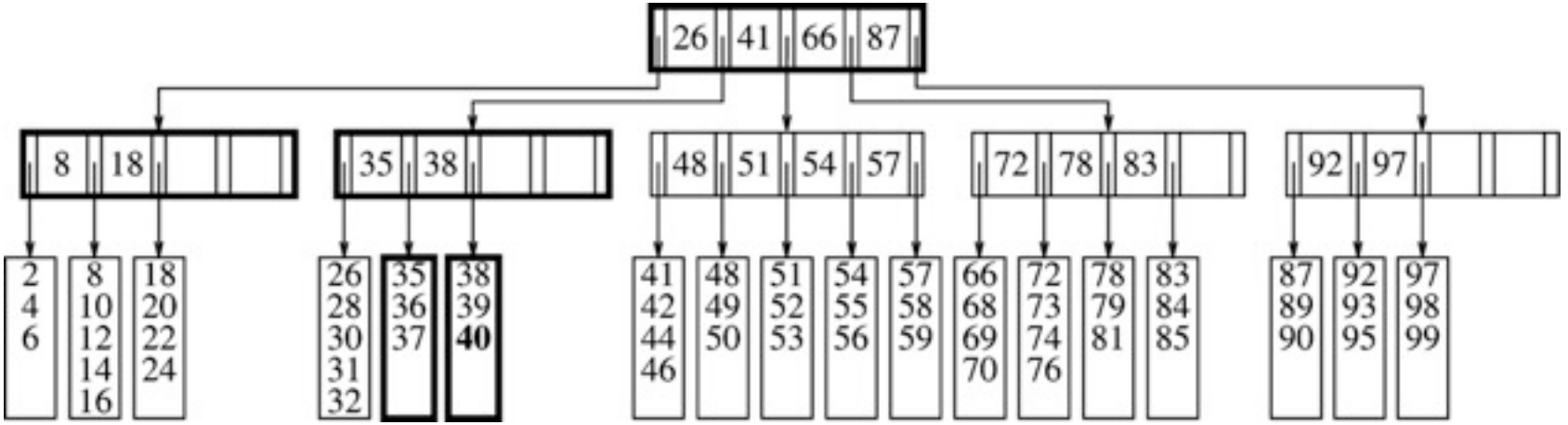
Insert 40 **Insert 29?**



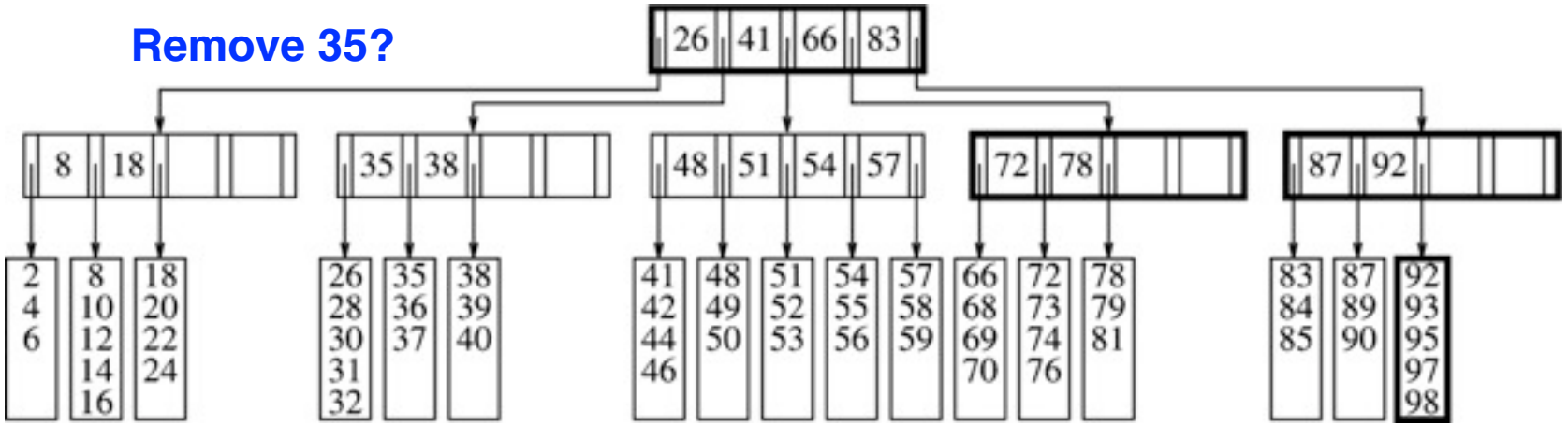
--

Deletion from B-trees: Node Merging

Remove 99:

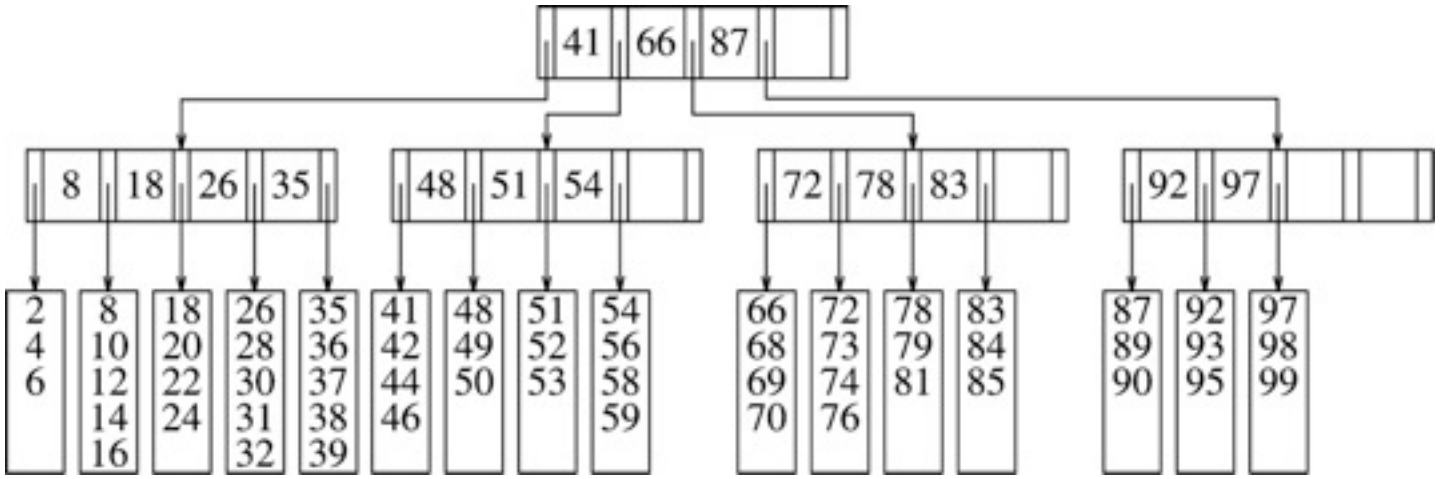


Remove 35?



Efficiency of B-Trees

- Often measured by number of disk accesses, instead of CPU comparisons
- How many disk accesses are required for each search/ insertion/deletion?
- CPU running time. Assume we use sorted array as the internal data structure of the nodes
  - for search?
  - for insertion/deletion?



Balanced or Unbalanced?

- Balanced search trees provide bounded time complexity
  - AVL trees
  - B-trees

  Many applications require quick access to the tree data structures, and balanced trees offer guaranteed performance

- In other applications, we may not care if the tree is balanced or not, as our objective is not to save CPU time

- Compression is an example (save disk storage space, save the size of network transfer, etc)

- Example: Huffman encoding

Fixed-length Character Encoding

- A character encoding maps each character to a number.
- Computers usually use fixed-length character encodings.
  - ASCII (American Standard Code for Information Interchange) uses 8 bits per character.

    A : 01000001

    B : 01000010

    C : 01000011

    …

    We can represent this encoding with a binary tree (height=8)

  - Unicode uses 16 bits per character to accommodate foreign-language characters. (ASCII codes are a subset.)
- Fixed-length encodings are simple, because
  - all character encodings have the same length
  - a given character always has the same encoding

Variable-length Character Encoding

- Problem: fixed-length encodings waste space.
- Solution: use a variable-length encoding.
  - use encodings of different lengths for different characters
  - assign shorter encodings to frequently occurring characters (the characters may have very different frequencies; this is a good property that we can capitalize on)

    E : 01

    O : 100

    S : 111

    T : 00

    TEST would be encoded as 000111100
- Challenge: when decoding/decompressing an encoded document, how do we determine the boundaries between characters?
  - example: for the above encoding, how do we know whether the next character is 2 bits or 3 bits?
- Solution: no character's encoding can be the prefix of another character's encoding (e.g., couldn't have T: 00 and Q: 001).

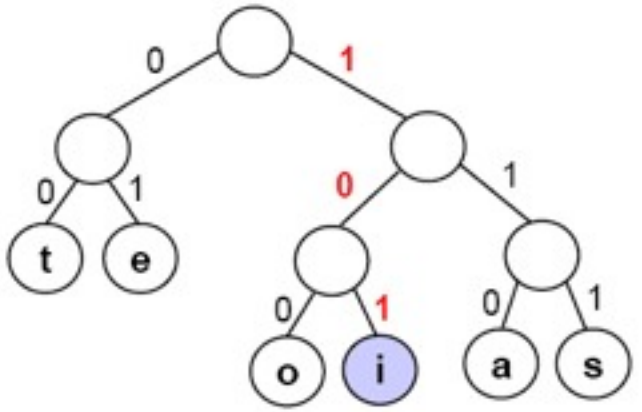
Huffman Encoding

- Huffman encoding is a type of variable-length encoding that is based on the character frequencies in a given document.
- Huffman encoding uses a binary tree:
  - to determine the encoding of each character
  - to decode an encoded file – i.e., to decompress a compressed file, putting it back into ASCII
- Example of a Huffman tree (for a text with only six chars):

  Leaf nodes are characters.

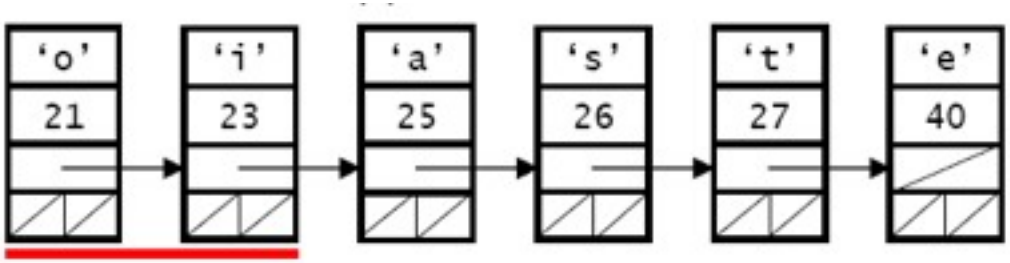
  Left branches are labeled with a 0,
  and right branches are labeled with a 1.

  If we follow a path from root to leaf,
  we get the encoding of the character in
  the leaf. An example: 101 = 'i'



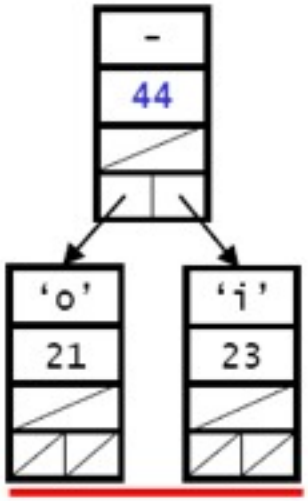
Building A Huffman Tree

- Begin by reading through the text to determine the frequencies.
- Create a list of nodes that contain (character, frequency) pairs for each character that appears in the text.



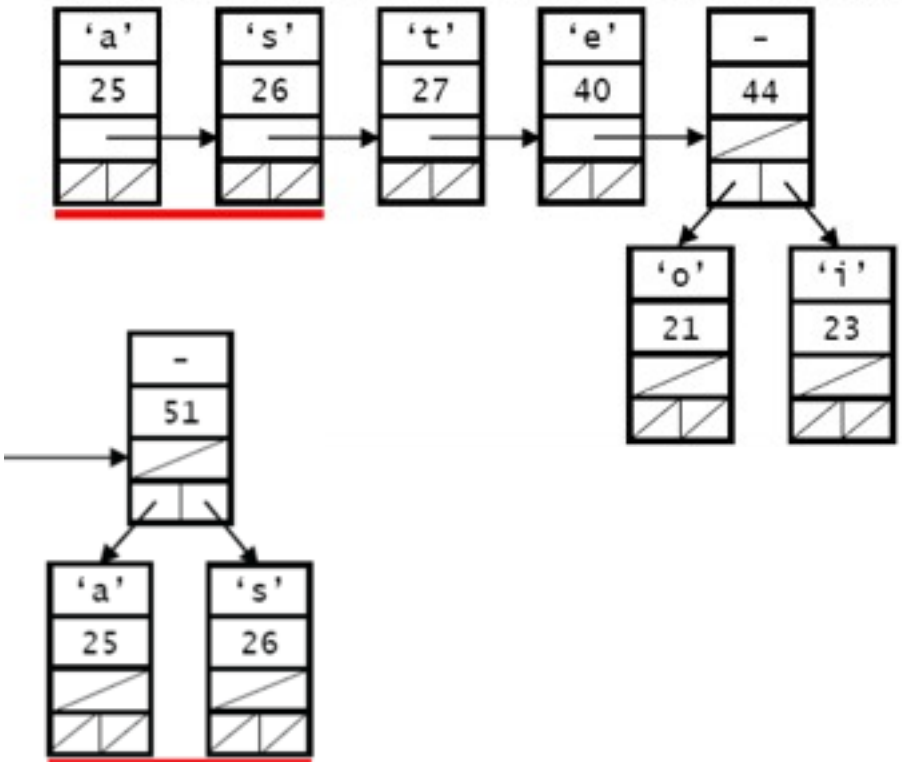
Remove and "merge" the nodes with the two lowest frequencies, forming a new node that is their parent.

- left child = lowest frequency node
- right child = the next lowest frequency node
- frequency of parent = sum of the frequencies of its children



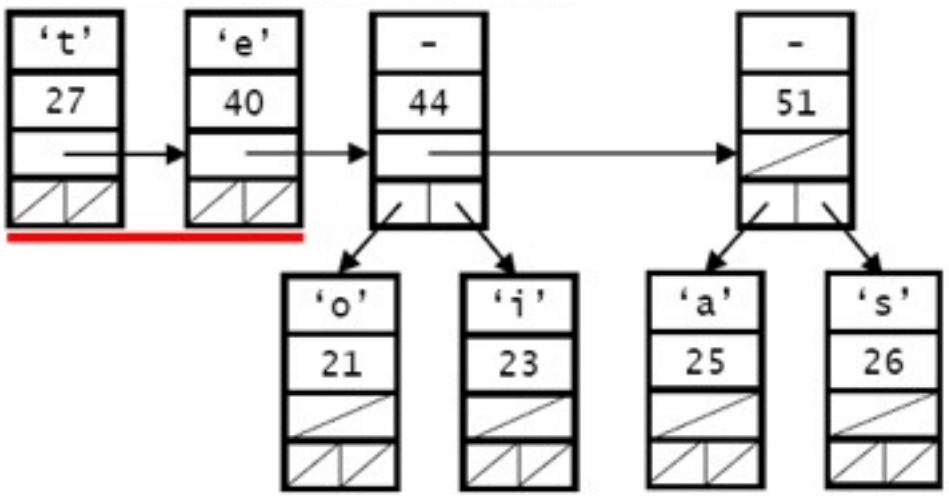
Building A Huffman Tree

- Merge the two remaining nodes with the lowest frequencies:
  - This is a greedy algorithms; a greedy algorithm attempts to get the optimal solution by finding currently the best solution in each step



Building A Huffman Tree

- It continues



Building A Huffman Tree

- The final tree, and character encoding