Other Kinds of Trees

CS240 Spring 2019

Trie Tree

Random Access Data

- Assume we have been given a task with the following requirements:
  - Store **volatile** data that is in a predefined range
    - i.e. constantly changing
  - Query data within a **known** number of steps
  - Queries should search entire data structure
    - so a Heap doesn't work

BST as a Solution

- A BST would be the best solution from the data structures we have seen so far
  - problems?
- For a BST, the value stored in the root node splits the range into two parts: less and greater than
  - Depending on the relationship between the root's value and overall the distribution of values, the resulting BST might be balanced or unbalanced.

BST Balancing Act

- The shape of a BST is determined by the order in which its data records are inserted.
  - One permutation of the records might yield a balanced tree while another might yield an unbalanced tree
    - balancing a tree is an expensive operation
- The BST is an example of a data structure based on **object space decomposition**
  - The current state of the tree and the values stored in the tree determine the future shape of the tree

What if we stored another way?

- The alternative is to use a predefined key range.
  - In other words, the root could be preselected to split the key range into two equal halves
    - Records with keys in the lower half of the key range will be stored in the left subtree,
    - Records with keys in the upper half of the key range will be stored in the right subtree.
- Splitting based on subdivisions of the key range is called **key space decomposition**.
  - This gives a better chance of a balanced tree

Key Space Decomposition

- Key Space Decomposition doesn’t guarantee a balanced tree
  - but at least the shape of the tree doesn’t depend on the order insertion.
- The depth of the tree is limited by the length of the key
  - If the keys are 3 characters long, the depth of the tree will be at most 3 levels
    - the depth of the tree can never be greater than the number of bits required to store a key value.

Key Based Trees

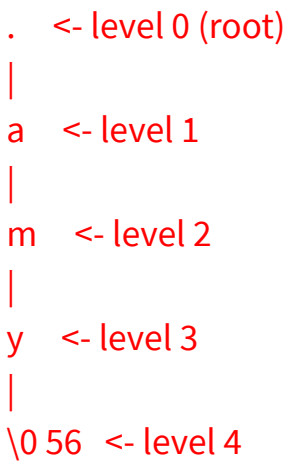
- Each item in the tree is required to have a unique key
  - BST has this same requirement, except the value is the key
- The key will determine the position in the tree
  - Each key will map to one path through the tree
- If the path is keys, where's the data?
  - The data must be in leaves, which means that data retrieval will, at best, take <minimum key size> and at worst <maximum key size>
- This kind of tree is called a **Trie**
  - Also called a re**trie**val tree

Trie Example

- Here is an example of a string based Trie
  - Suppose we want to store a bunch of name/age pairs for a set of people (assume the names are unique).
    - Here are some pairs:
      - *amy 56*
      - *ann 15*
      - *emma 30*
      - *rob 27*
      - *roger 52*
  - Now, how will we store these name/value pairs in a trie?
    - A trie allows us to share prefixes that are common among keys. Again, our keys are names, which are strings.

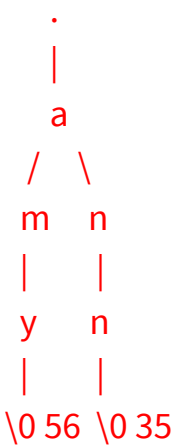
Trie Example

- Let's start off with amy.
  - Build a tree with each character in a separate node.
    - Note the root must just be an entry point
  - We need one node under the last character in her name.
    - In the final leaf node, we'll put the null character (\0) to represent the end of the name.
    - If the keys are of variable length, **we must have some delimiter,** such as a null byte, to denote when we have found our leaf
      - *This last node is also a good place to store the age for amy.*
- Each level in the trie holds a certain character.
  - The first character of a string key in the trie is always at level 1 (not 0), the second character at level 2, etc



Add New Values to the Trie

- Now, when we go to add ann, we do the same thing;
  - we already have stored the letter ‘a’ at level 1, so we don't need to store it again,
    - we can just reuse that node with a as the first character.
  - Under a (at level 1), however, there is only a second character of m, but, since ann has a second character of n, we'll have to add a new branch for the rest of ann
    - The branches still should be ordered, so n is to the right of m
- Again, ann's data (an age of 35) is stored in her last node.

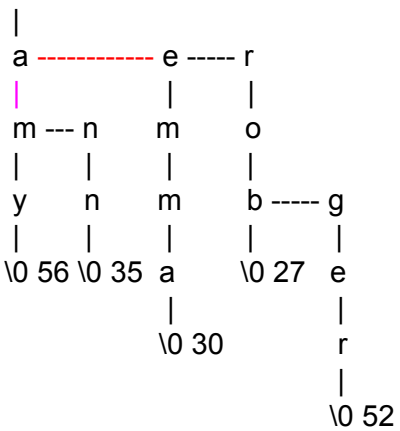


Sub-Key Values

- What would happen if we now try to add:
  - Ethan, 24
  - Emma, 17
  - Em, 30
- Would the tri look different if we added them in different order?
  - No, Tries with equal values are identical, regardless of insertion order
- Is this a binary tree, tertiary tree or what?
  - In other words, each node has at most how many children?

Structure

- What does the class for the Trie node look like?
  - There are two implementations
    - class Node{
      public:
      Node children[<keysize>] //this implementation can be a problem because it uses
      <type> key; //a lot more memory
      <type> * data;
      };
    - class Node{
      public:
      Node * next, * child; //increases lookup time
      <type> key;
      <type> * data;
      };



Trie Summary

- Features:
  - Uses keys rather than object values to sort data into a tree structure
  - All data is stored in leaf nodes
  - Every Trie with the same data will have the same structure regardless of insertion order
- Uses:
  - Word validation or lookup
    - For example, what is the most efficient way to find if a word exists in a long block of text?
      - *read all words into a trie, then lookup becomes O(n) where n= longest word*

Trie wastes memory

- If the actual data is only stored in a leaf node, then all internal nodes that have data location are wasting space
- Solution?
  - Two separate node types:
    - An internal node that only contains a key and pointers to children/siblings
    - A leaf node type that only contains key and data
  - Both would derive from a parent node class
- The two node types solution adds more complexity to your code, more chances for errors

Clumping

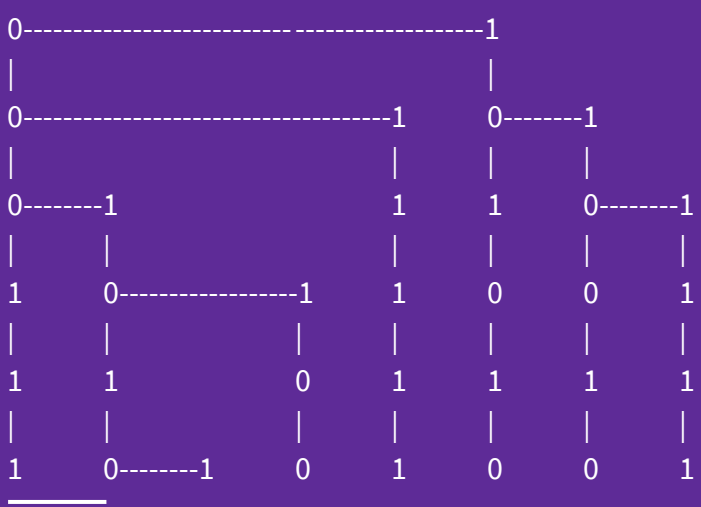
- What if we want to use a trie to store words in the english language
  - How many words are going to be down the **z** and **q** branch?
  - How many words are going to be down the **s** and **r** branch?
- What problem does this reveal
  - Data clumping is common with real world data
  - The linked sibling solution helps with memory usage, but we will still end up with an unbalanced tree

Classwork

Binary Trie

42: 101010
12: 001100
10: 001010
50: 110010
31: 011111
7: 000111
11: 001011
63: 111111

8 bits * 6 = 48 bits, stored in 34 bits



B-Tree

https://www.cs.usfca.edu/~galles/visualization/BTree.html

Disk Storage Problem

- Most data structures are designed to balance memory usage with quick access to specific data
- What if all our data doesn’t fit into main memory? In other words, our focus is on minimizing disk access, rather than finding specific data.
    - Disk access is one of the slowest operations on your machine
    - So, with each disk read, read the largest block of data you can to minimize reads

Possible Solutions

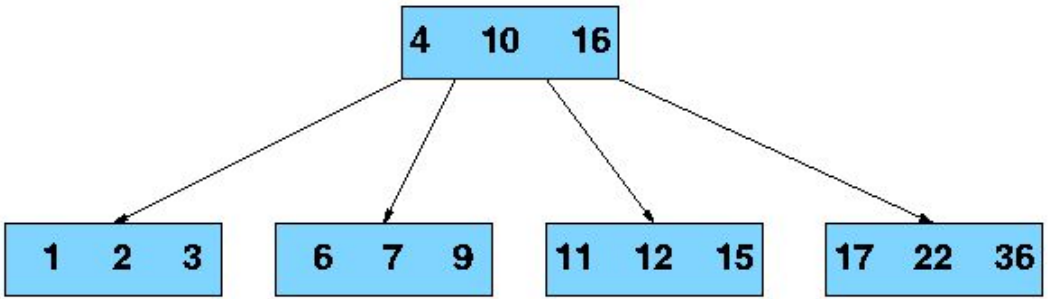
- Binary search tree cannot solve our problem because files are constantly getting created and deleted
  - Would require constant rebalancing
- In a Trie, do not know if an entry is in the tree, until we make it all the way to the leaf node
  - Our worst case is a search requiring the length of the key
    - In the case of memory lookup, the key would most likely be a memory address, which on most machines would be 64 lookups just to find a file
- Both of these solutions also only access a single element

B-Tree

- Our solution should optimize for disk blocks (groups of data), not specific data
- A B-tree is a balanced tree where:
  - All leaves are at the same level
  - Every node has a maximum number of children, ***t***, and minimum ***t/2***
  - Every node, except root, must have at least **t-1** keys
    - So non-leaf nodes always have 1 more child than the number of keys
  - Nodes may contain, at most, **2t-1** keys
  - Keys are sorted in increasing order
  - Grows upwards toward the root, rather than downward like a bst

Example

- The below B-Tree is an m=4 way search tree.
  - each node has a
    - maximum of 3 keys and 4 ‘links’
    - minimum of 1 key and 2 ‘links’
  - Allows you to store up to 15 values with 1 traversal
- A combination of arrays and links for maximum efficiency



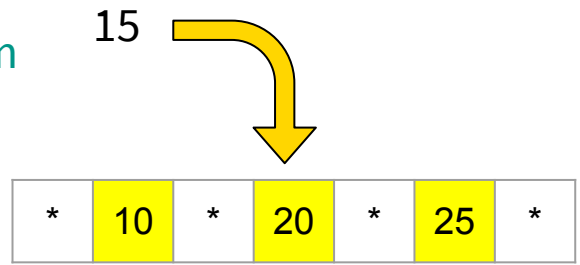
Inserting into a B Tree

Insertion

- The below B-Tree is an m=4 way search tree.
  - After inserting the following values:
    - 20, 10, 25
- Notice the node is kept sorted.
  - Before we insert an element, we check to make sure there is enough room
  - We insert in sorted order

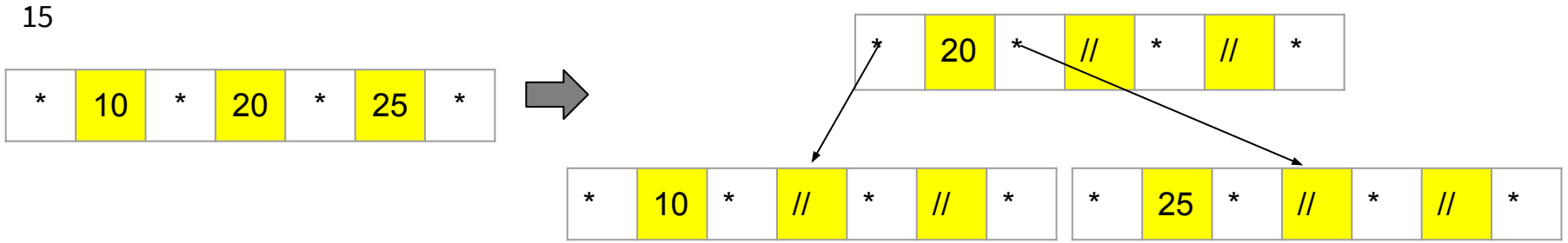
Insertion

- Inserting 15 means we have run out of space, and will need to reorganize our tree
  - There are two different methods for insertion, proactive and reactive
    - **proactive** adjusts the tree first, then adds the new element
    - **reactive** adds the element, then adjusts the tree
      - *generally, the proactive approach is more efficient because it requires less traversal*
  - We always traverse to insert at the insert at the leaves
    - since we only have a single node, our root is a leaf
  - To make space, we use the splitChild() algorithm



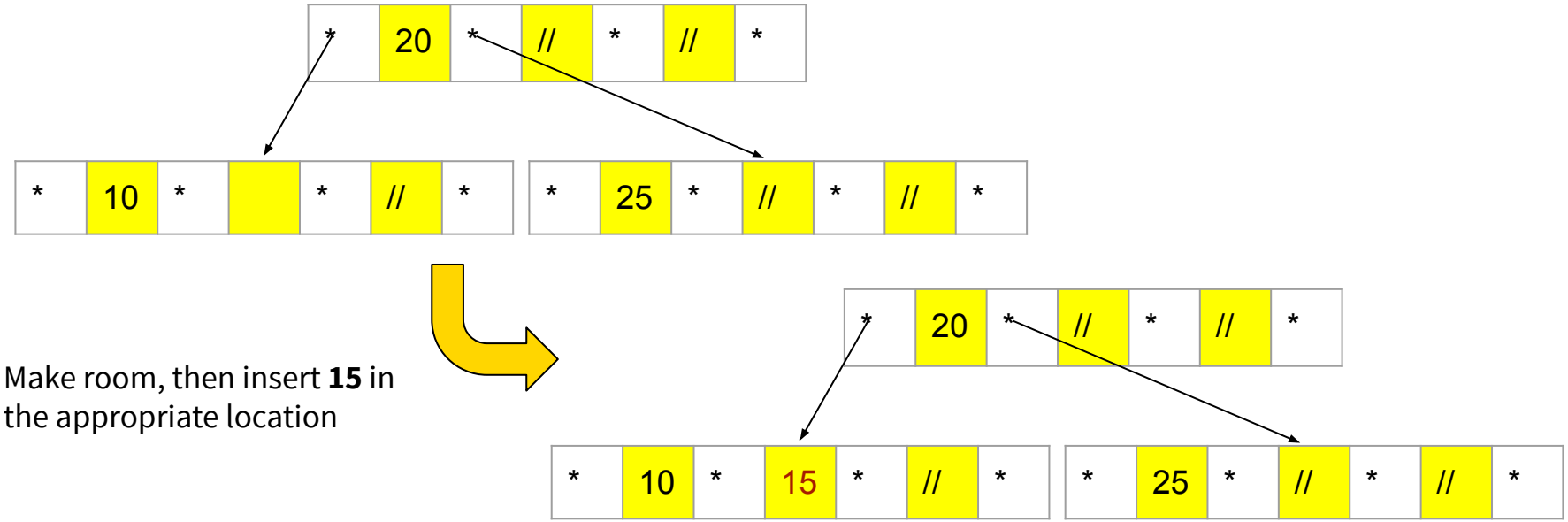
SplitChild()

- If leaf node is full, split on the middle value



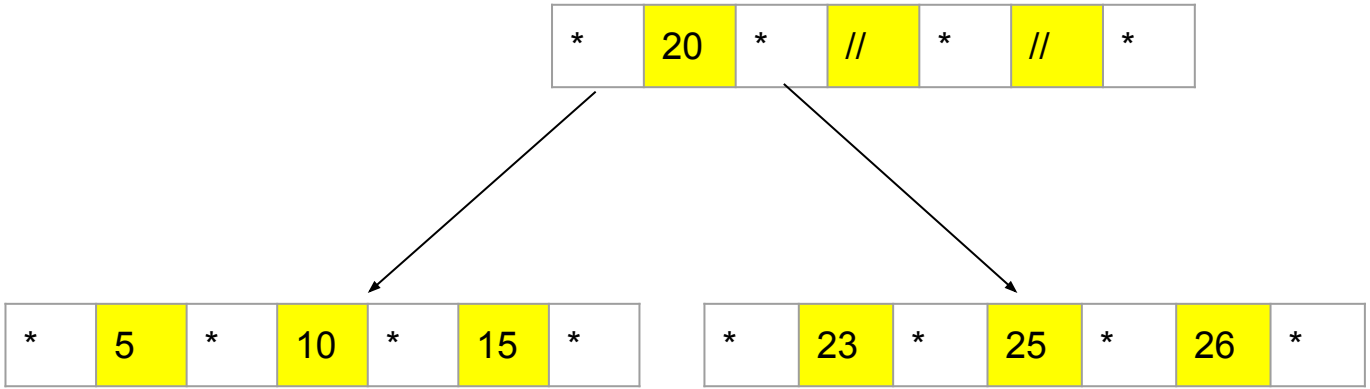
SplitChild()

- If leaf node is full, split on the middle value



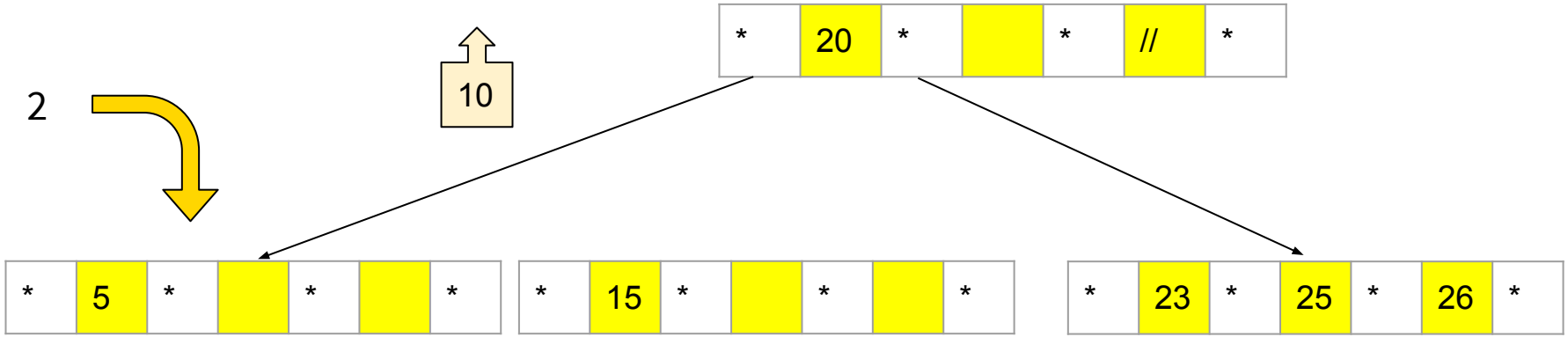
Insertion

- We can now insert 5, 23, and 26 without any alteration because the subnodes have room to accept values
  - Every node must remain sorted, but this isn't a problem. Why?



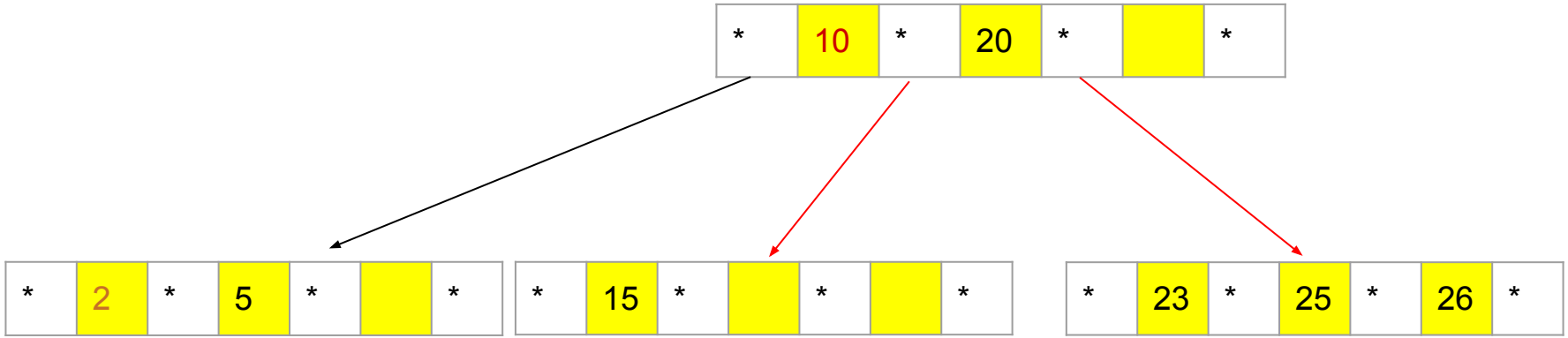
Insertion

- We now want to insert 2 into the BTree
  - We will have to split the node
    - Choose the median value (which should be the middle value) as the boundary key
      - *The boundary key will be removed and, eventually, promoted*
  - We must then reorganize our parent node



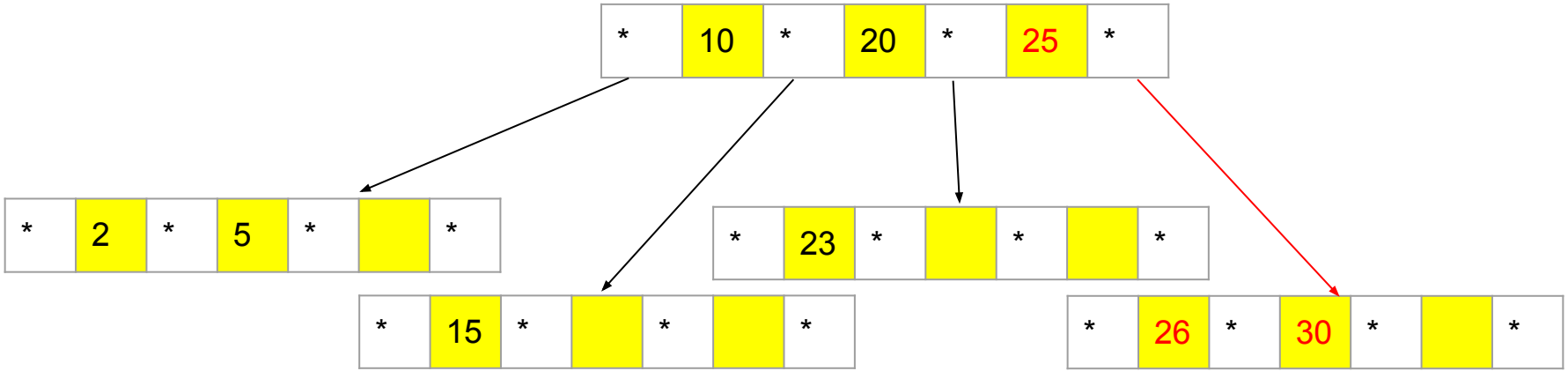
Insertion

- Insert 10 into the parent in sorted order
- Adjust children positions
  - This will, at most, require shifting links over



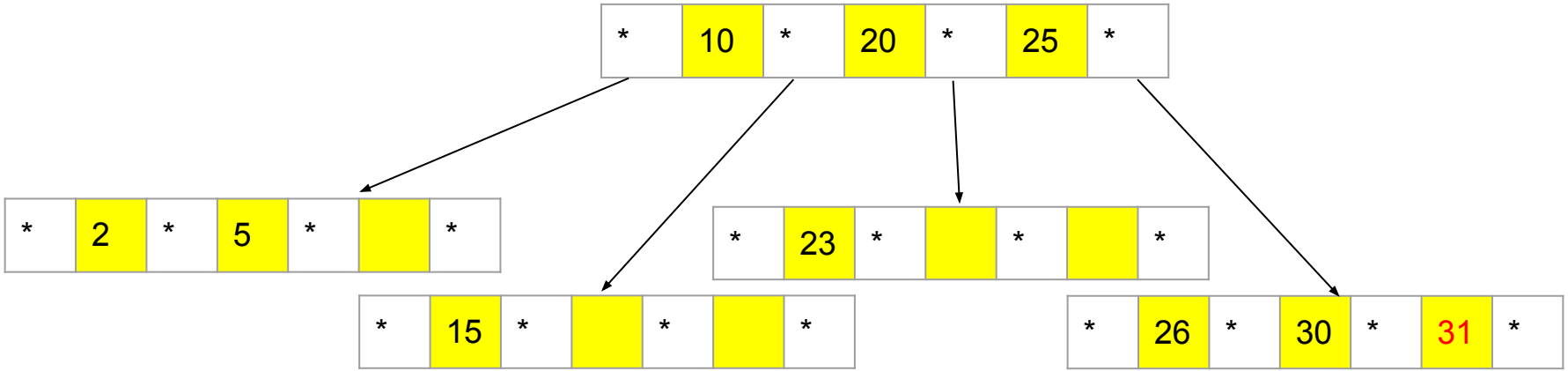
Full Parent Node - Part 1

- If we add 30, our parent node becomes full.



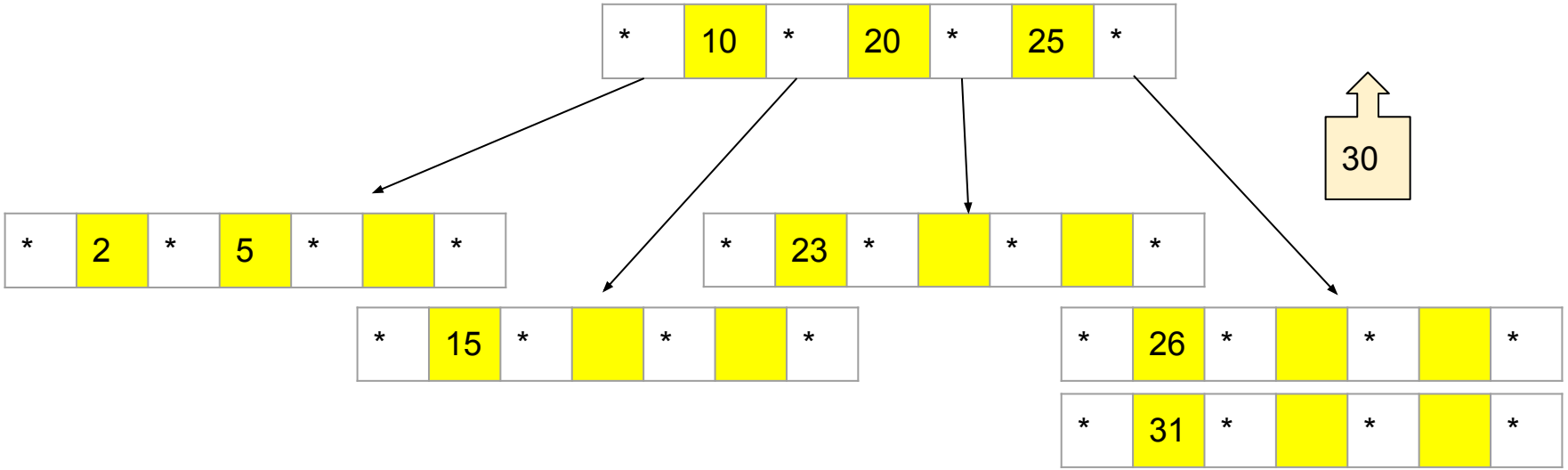
Full Parent Node - Part 1

- Lets, add 31 and 32
  - 31 requires no restructuring
  - 32 requires splitChild()

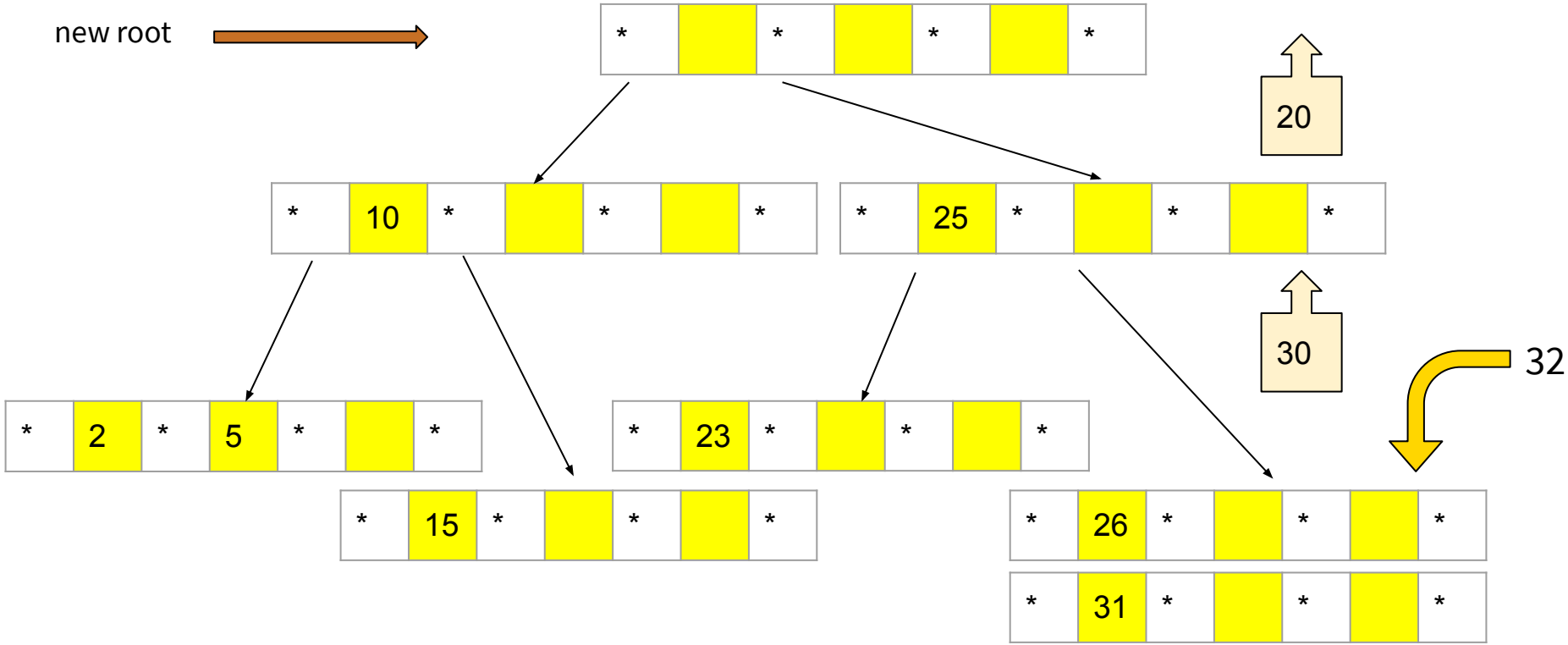


Full Parent Node - Part 1

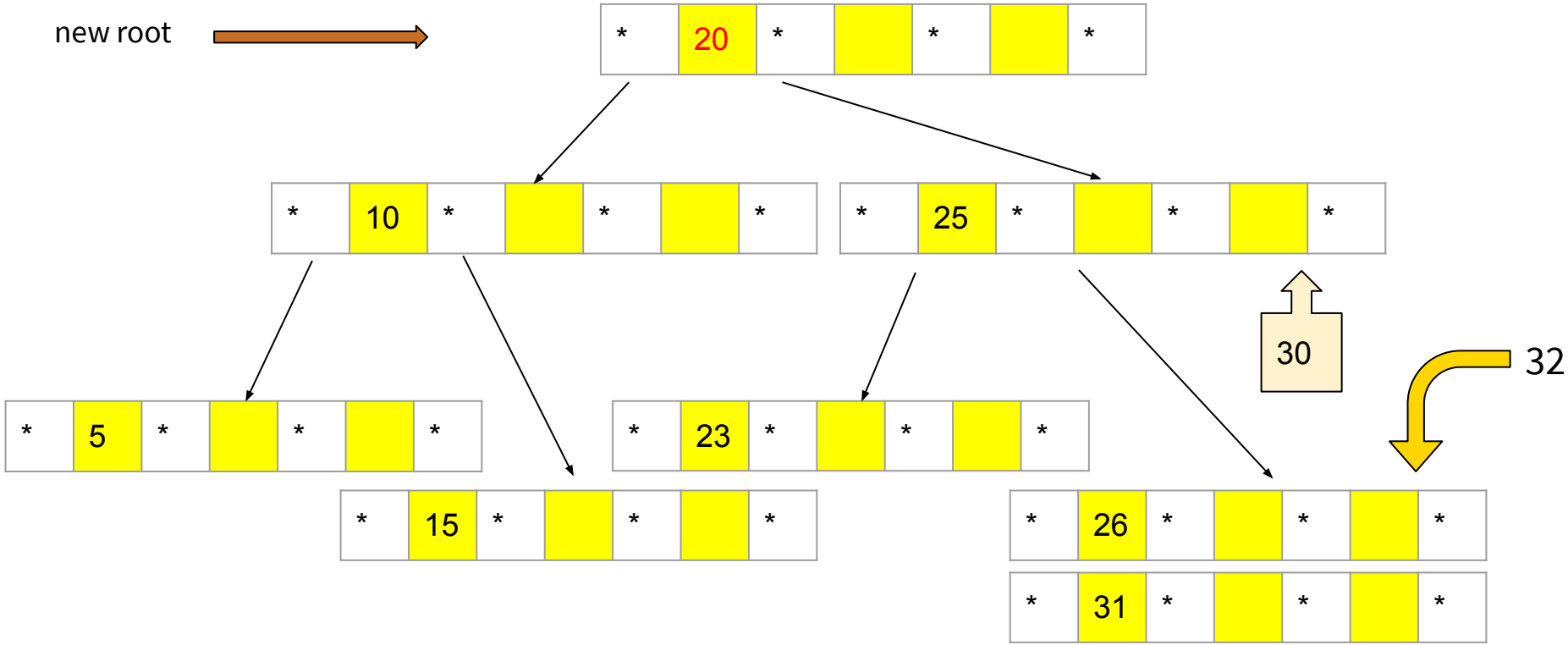
- We need to promote 30, but the root is full
- perform the same splitChild again



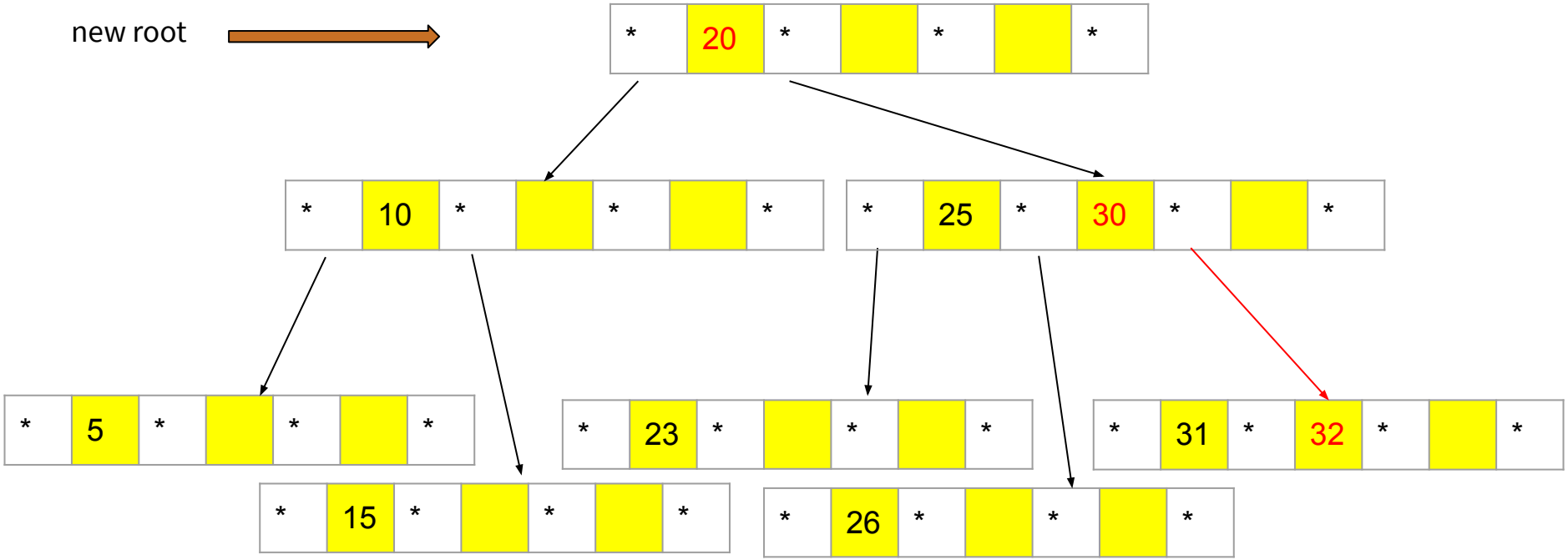
Full Parent Node - Part 1



Full Parent Node - Part 1



Full Parent Node - Part 1



Why are B-Trees used for Data Storage?

- Update and search operations affect only a few disk blocks.
  - The fewer the number of disk blocks affected, the less disk I/O is required.
  - The size of a node (number of value/keys), can be mapped to the size of a single disk block
- B-trees guarantee that every node in the tree will be full to a minimum percentage.
  - This improves space efficiency while reducing the typical number of disk fetches necessary during a search or update operation.

B Tree Problems

- The B tree concept is universally used to store large file
  - But it isn't very space efficient
    - Key also contain objects, so internal nodes store large chunks of memory
    - This means that the entire tree structure cannot fit into main memory
      - *Wait, wasn't that the problem the btree was solving?*
- The B tree requires full tree traversal to fully scan all data
  - This is a more memory intensive operation

B+ Tree

- Most systems use a variant called the B+ tree:
  - Only stores data in the leaf nodes
  - Internal nodes only store keys
    - Keys are just path guides
    - Entire tree structure - leaves can be stored in main memory
  - Leaf nodes are linked for traversal
- B+Tree allows better space efficiency because you do not need to store entire records in internal nodes, just the keys
- B+Trees also allow easier traversal of all records

Classwork

B Tree