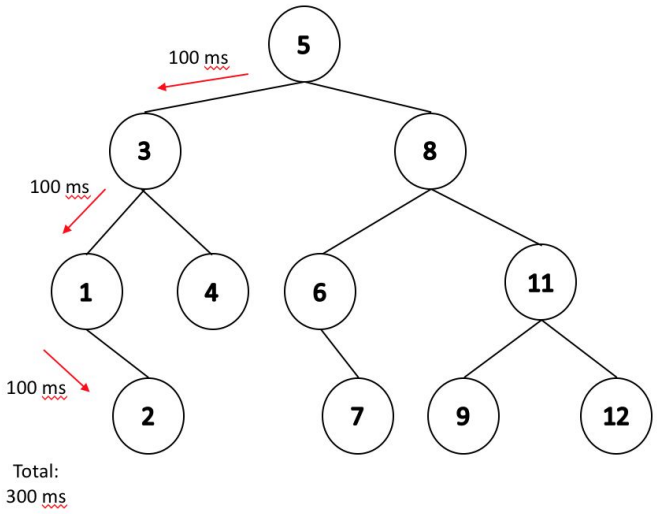
CS 225 Spring 2019 :: TA Lecture Notes
3/11 B Tree + B Tree Analysis

By Wenjie

- **B Tree Motivation**
  - We cannot always keep data in the memory
  - Runtime
    - We assume all operations have uniform runtime, but this is not true
    - If tree is not stored in memory, every tree access takes some time
    - However, seeking data from disk may take 40ms+. An O(lg(n)) AVL tree no longer looks great.



- **BTree (of order m)**
  - Goal: we want to minimize the number of reads.
  - In practice, every node will store exactly (1 network packet/1 disk block/...)
  - BTree node:
    - Every node is a sorted array
    - Contains up to m - 1 keys

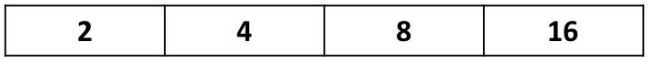
-3	8	23	25	31	42	43	55

m = 9

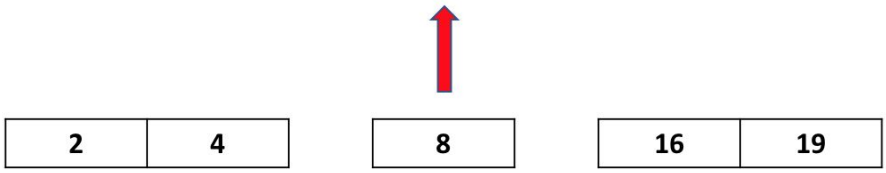
  - Insertion (m=5)
    - We add to the tree until we reach the maximum number of keys in a node.

CS 225 Spring 2019 :: TA Lecture Notes 3/11 B Tree + B Tree Analysis

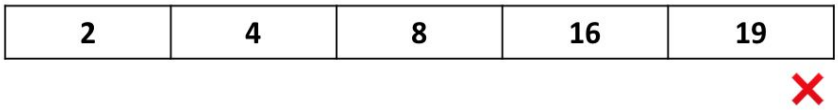
By Wenjie

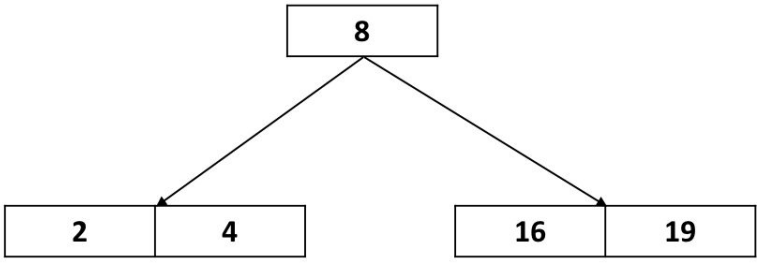


- If we insert for example 19, we will exceed the allowed number of keys in this node. This is a overfilled array
- we split the data and throw up the middle element.

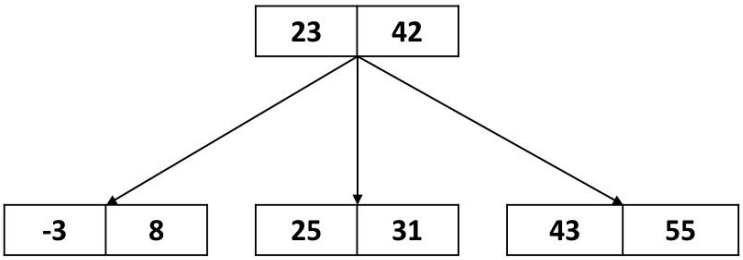


- Now we have a BTree with order m=5





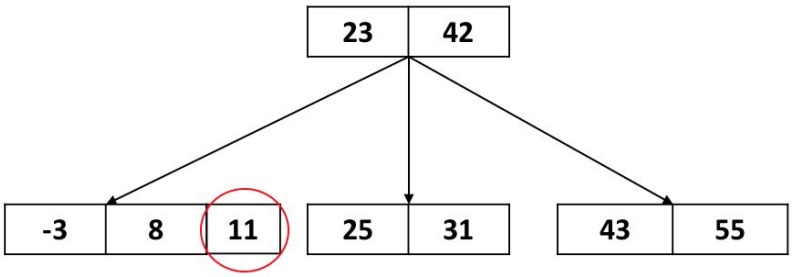
- Recursive call of split
  - m = 3



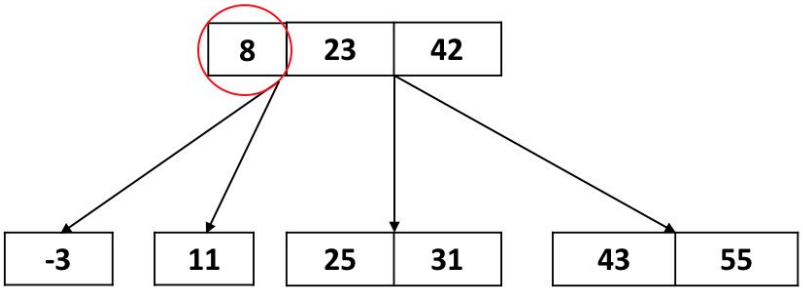
CS 225 Spring 2019 :: TA Lecture Notes 3/11 B Tree + B Tree Analysis

By Wenjie

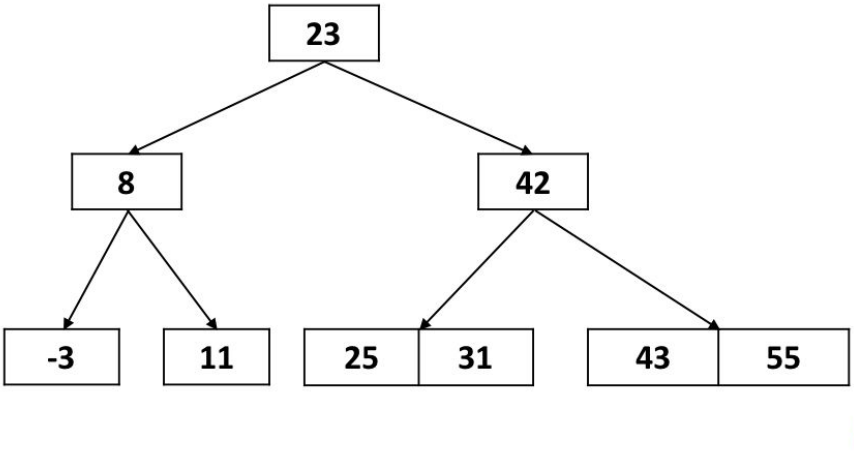
- Insert 11



- At the leftmost child node, we have an overfilled array. Therefore, we need to split data and throw up key number 8.



- We fixed the child node, but now the parent is overfilled. Thereby, we split and throw up again.



✅

- Finally, we have a valid BTree of order 3.

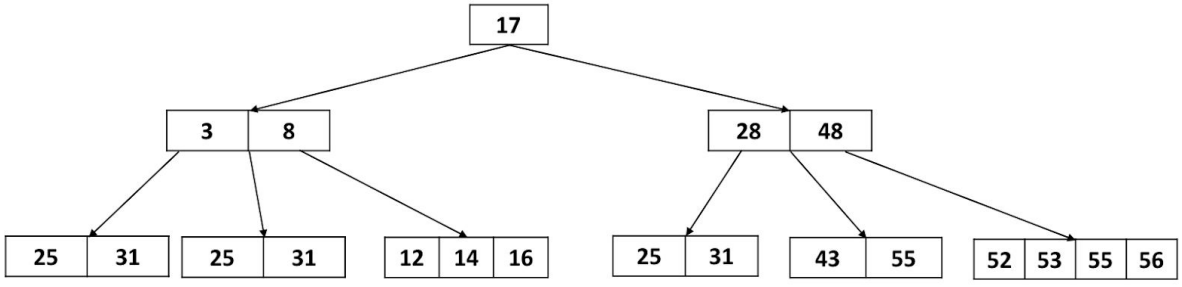
CS 225 Spring 2019 :: TA Lecture Notes
3/11 B Tree + B Tree Analysis

By Wenjie

- **BTree structure**
  - Everything on the left subtree is less than the left key.
  - Everything on the right subtree is greater than the right key.
  - Middle tree is the data in between left and right key.
- The tree only gets taller when the leaf node is overfilled, and its parent is overfilled, and its parent's parent is overfilled...and also the root is overfilled...

- **BTree properties**
  - Basics
    - A BTree of order **m** is an m-way tree.
    - All keys within a node are ordered.
    - All leaves contain no more than m - 1 keys.
  - Number of children/keys
    - All internal nodes have exactly one more child than keys.
    - Root node can be a leaf or have [2, m] children, [1,m] keys.
    - All non-root nodes have [ceiling(m/2), m] children.
    - All non-root nodes have [ceiling(m/2)-1, m-1] keys.
  - All leaves are on the same level.
    - always a perfect tree!

- **Determining the order of a BTree**
  - Look at the children range. Number of children should be between [ceil(m/2), m].
  - Eg:



    - Lower bound: Look at the node having the most children
      - m >= 5

CS 225 Spring 2019 :: TA Lecture Notes
3/11 B Tree + B Tree Analysis

By Wenjie

        - Upper bound: Look at the node having the least children
            - ceil(m/2) <= 3, m<= 6
        - Therefore m = 5 or 6
        - **Usually, m** is odd. So *probably* 5, and things work out better that way.

- **BTree Search**
    - Search through the array
        - If the data is not in memory linear search is better because we can search while reading-in the data.
        - If the data is in memory, we should do binary search.
        - However, this running time is not physical and in the function we are bounded by the physical time (this means that the running time at this point in code is not the most important).
    - Check for match.
    - Recursive search
        - If is leaf, we cannot find the element
        - If not leaf, go deeper.
        - this is the bottleneck of the search. Fetch child is slow because of the physical time required (disk time, internet lag).

BTree.cpp

```
6  bool Btree::_exists(BTreeNode & node, const K & key) {
7      unsigned i;
8      for ( i = 0; i < node.keys_ct_ && key < node.keys_[i]; i++) { }
9
10     if ( i < node.keys_ct_ && key == node.keys_[i] ) {
11         return true;
12     }
13     if ( node.isLeaf() ) {
14         return false;
15     } else {
16         BTreeNode nextChild = node._fetchChild(i);
17         return _exists(nextChild, key);
18     }
19 }
```

CS 225 Spring 2019 :: TA Lecture Notes
3/11 B Tree + B Tree Analysis

By Wenjie

B Tree Analysis

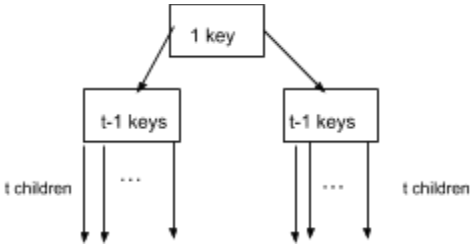
In our AVL Analysis, we saw finding an upper bound on the height (given n) is the same as finding a lower bound on the nodes (given h).
And here we want to find a relationship between the number of keys **n** and the height of the BTree **h**. In other words: how is the height **h** bounded by the keys **n?**

Minimum number of keys in a BTree of height **h** and order **m**:

We can do a Level by level analysis:

Let t= ceil(m/2)

Level	Nodes	Keys	Children
Root	1	1	2
1	2	2*(t-1)	2*t
2	2*t	2*t*(t-1)	2*t^2
3	2*t^2	2*t^2*(t-1)	2*t^3
...	...	....	....
h	2*t^(h-1)	2*t^(h-1)*(t-1)	0 (leaves)



The sum of all level gives:

minimum number of keys: $2t^h - 1$

minimum number of nodes: $2 \cdot \frac{t^h - 1}{t-1} + 1$

Min total nodes = $1 + 2 + 2 * t + 2 * t^2 \ldots + 2 * t^{(h-1)} = 1 + 2 * \sum_{i=0}^{h-1} t^i = 1 + 2 * \frac{t^{h-1+1}-1}{t-1} = 1 + 2 * \frac{t^h - 1}{t-1}$

Min total keys = $1 + 2 * \frac{t^h - 1}{t-1} * (t-1) = 2 * ceiling\,(m/2)^k - 1 = 2 * t^h - 1$

Root has 1 key

All other nodes have >= t-1 keys

CS 225 Spring 2019 :: TA Lecture Notes 3/11 B Tree + B Tree Analysis

By Wenjie

Thus: $n \geq 2 * t^h - 1$ *(for any BTree of height h and order m)*

Solving for h: $\frac{n+1}{2} \geq t^h \Rightarrow log_t(\frac{n+1}{2}) \geq h$

Since $t = ceil(m/2)$ we can say: $log_t(\frac{n+1}{2}) \sim log_m(n)$

Thus we have: $h \leq log_m(n) \Rightarrow seeks \leq log_m(n)$

Given m=101 , a BTree of height h=4 has:

Minimum keys: $2 * t^h - 1 = 2 * ceil(101/2)^4 - 1 = 2 * (51)^4 \approx 12.5\ million$

Maximum keys: //Practice problem