1. B- Trees:

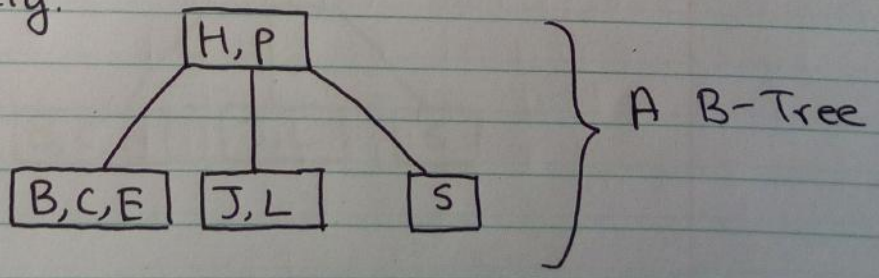
- Properties:
  - A self-balancing tree.
  - All leaves are at the same level.
  - A B-tree is defined by a min degree, t. For our purposes, $t=2$. This is known as a (2, 4) tree or 2-3-4 tree.

  - A B-tree's nodes can have c children and $c-1$ keys where $t \leq c \leq 2t$. I.e. A node can have at most $2t-1$ keys.

Note: Since $t=2$, each node, except for the root, must have 2 to 4 children. All nodes must have 1 to 3 keys.

Note: The root is the exception. For the root, c can be less than t. The root must have at least 1 key, and no more than $2t-1$ keys.

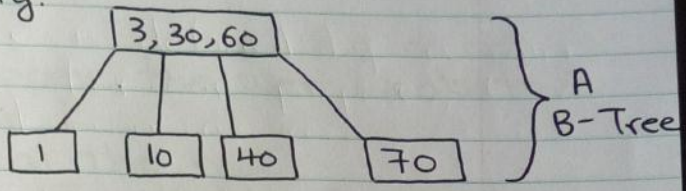
  - All keys are in ascending order. I.e $k_1 < k_2 \ldots < k_{c-1}$
  - $k_{i-1} <$ (keys in child subtree T_i) $< k_i$, if k_{i-1} and/or k_i exist.
  - E.g.



A B-tree needs to have btwn $t-1$ to $2t-1$ keys per block.

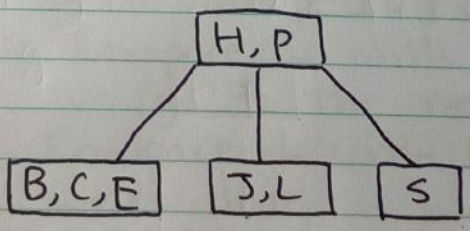
Here, [H,P] is the root.
Note the following:
1. All keys are ordered in ascending order.
2. B, C, E < H
3. H < J, L < P
4. P < S

- E.g.



Note that:
1. 1 < 3
2. 3 < 10 < 30
3. 30 < 40 < 60
4. 60 < 70

- Search:
  - Start at root r of T.
  - Go through the keys of r until k is found or current key k_i is the rightmost key or $k < k_i$.
  - If $k < k_i$, recursively call search(T_i, k).
  - Otherwise, recursively call search(T_{i+1}, k).
  - E.g. Consider the B-Tree below.



Suppose the user wants to search for P.
First, we go to the root and go to H. Since P>H, we go to the next key.
The next key is P, which is what the user is looking for, so we're done.

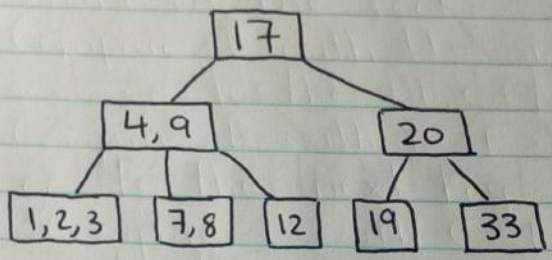
Next, suppose the user wants to search for E.

1. Start at the root.
2. Compare E with H. Since E<H, we go to the [B, C, E] node.
3. Compare E with B. E>B, so we go to the next key.
4. Compare E with C. E>C, so we go to the next key.
5. Compare E with E. E=E, so we are done.

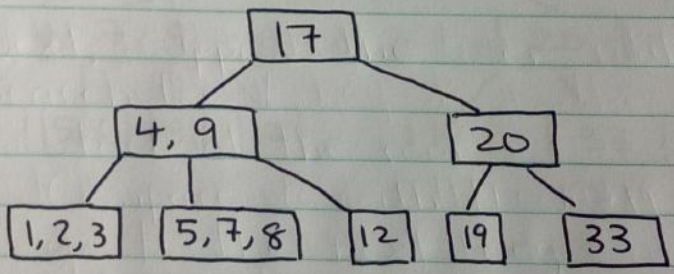
Next, suppose the user wants to search for L.

1. Start at the root.
2. Compare L with H. L>H, so we go to the next key.
3. Compare L with P. L<P, so we go to the [J, L] node.
4. Compare L with J. Since L>J, we go to the next key.
5. Compare L with L. L=L, so we're done.

- Insert(T, k, v):
  - First, we traverse the tree to find the appropriate leaf to insert k.
  - If the leaf is not full, we simply add the key.
  - If the leaf is full, inserting the key causes an overflow.
  - E.g. Consider the B-Tree below.



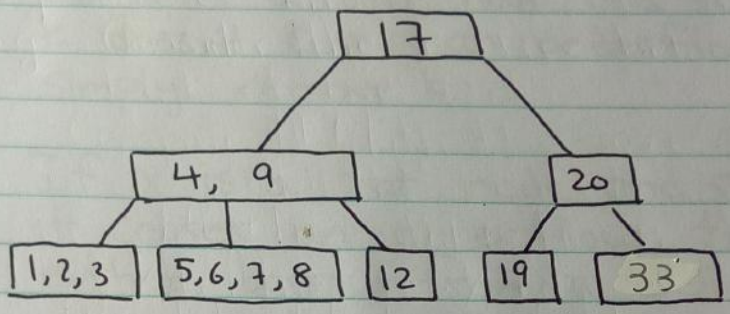
First, let's insert 5. The B-Tree looks like this, now.



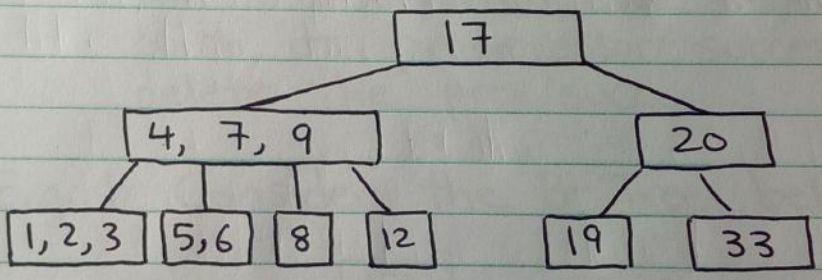
If we insert 6, we cause an overflow. Each node can obtain at $2t-1$ keys, and since $t=2$, this means that each node can have at most 3 keys. Inserting 6 will cause a node to have 4 keys.

To correct this, we insert 6 and select the median of the four keys, rounded up, and put that in the parent node. If that causes an overflow in the parent node, you repeat the process until there are no more overflows.

Insert 6:



Push 7 to the parent node.



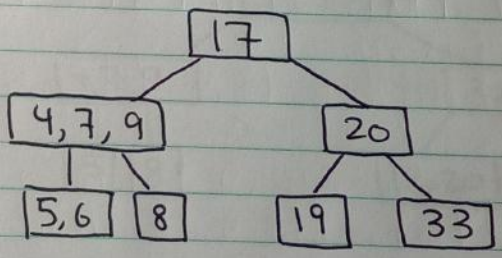
- Delete(T, k):
  - Deleting a key may cause an underflow because there might not be at least t children.
  - The steps for delete are as follows:
    1. Search for k.
    2. Let x be the node that should contain or contains k.
    3. If x is a leaf node and removing k doesn't cause an underflow, we simply remove k.

       If x is a leaf node and removing it causes an underflow, then either borrow from a sibling or merge with a sibling.

       If x is internal, we replace k with its predecessor/successor and delete the pred/succ.

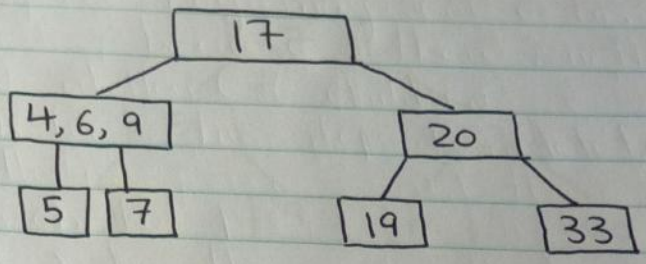
Note: Another cause of underflow is when a block does not have at least 1 key.

- E.g. 1 Consider the B-Tree below.

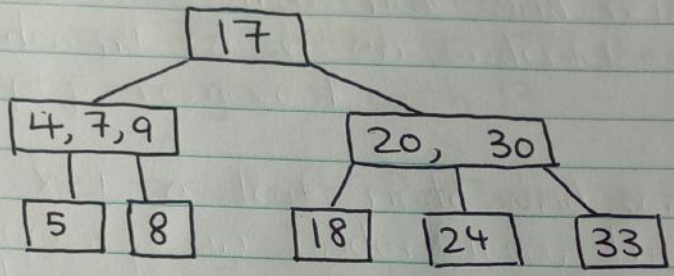


If we delete 8, then [4,7,9] will only have 1 children. To fix this, we can borrow from a sibling. [5,6] can spare a key, so we push 6 up to the parent node and push 7 down.

The result B-Tree is this:

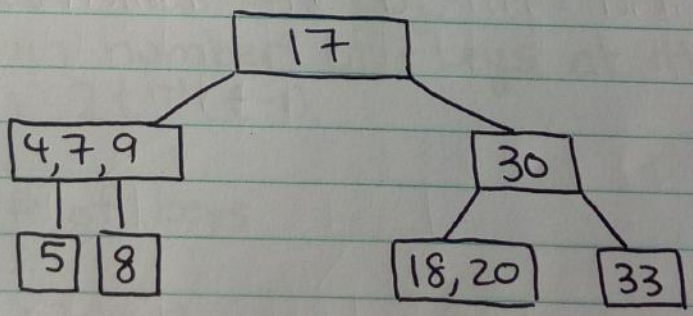


- E.g. 2 Consider the B-Tree below.

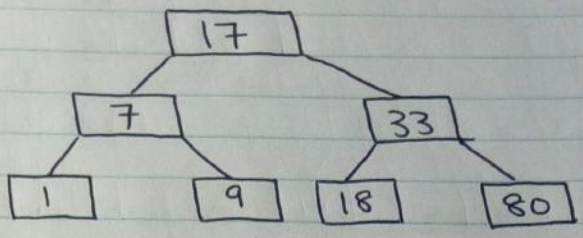


Consider removing the node 24. Neither 18 nor 33 can spare a key, so we merge with 1 of the siblings, say 18.

The resultant B-Tree is this:

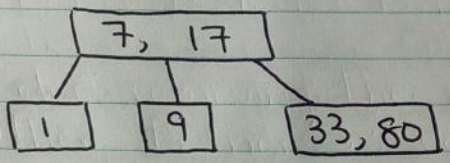


- E.g. Consider the B-Tree below.



Suppose we want to delete 18. We can't borrow from 80 nor merge 33 with 80. Instead, we merge 7 with 17.

The resultant B-Tree is this:



- B-Tree Height:
  - Suppose that the tree is non-empty, so $n \geq 1$.
  - The root has ≥ 1 key.
  - The min number of keys at the i^{th} level is $2t^{i-1}(t-1)$.

Depth	# of keys
0	≥ 1
1	$\geq 2(t-1)$
2	$\geq 2t(t-1)$
3	$\geq 2t^2(t-1)$
⋮	⋮
i	$\geq 2t^{i-1}(t-1)$

- Therefore, the number of nodes is at least:

$$n \geq 1 + \sum_{i=1}^{h} 2t^{i-1}(t-1)$$

$$= 2t^h + 1$$

and the height is at least $\log_t\left(\frac{n+1}{2}\right) \geq h$.

- B-Tree Time Costs:
  - Find, insert and delete each reads/writes $\Theta(\log_t(n))$ nodes.
  - Note: The constant multiplier is around 1 to 3.
  - We choose t s.t. one node fits in 1 block. That way, search, insert and delete each writes/reads $\Theta(\log_t(n))$ blocks.
  - t is usually in the thousands.
  - E.g. $\log_{1024} n = \frac{\lg n}{\lg 1024}$

$$= \frac{\lg n}{10}$$

  - This is better than binary search trees' $\lg n$.

2. Bloom Filter

- Definition:
  - A bloom filter is a space-efficient probabilistic data structure that is used to test whether or not an element is a member of the set.
  - The price we pay for efficiency is that it is probabilistic in nature. This means that there might be some false positives.
    - A false positive means that the bloom filter says an element is in a set, when it is not.
    - Bloom filters never generate false negatives. If the bloom filter says that an element is not in a set, then the element is not in the set.
- Operations:
  - Let S be a dynamic set of keys drawn from a universe U. A bf maintains a summary F_S of S, supporting the following operations:

1. Insert(F_S, x): This adds x to S. I.e. $S = S \cup \{x\}$

2. Search(F_S, x): This searches for x in S. If $x \notin S$, then this returns "no." If there is a high enough probability that x is in S, this returns "yes."

Note: There is no delete operation.

- How BF's Work:
  - A BF consists of an array of m bits, initally all set to 0. Let $h_1, h_2, \ldots h_t$ be hash functions that map U to $\{0, 1, \ldots m-1\}$. The BF operations are then implemented as follows:
    - To insert a key x into the BF, we set all the bits $BF[h_1(x)], \ldots, BF[h_t(x)]$ to 1.
    - To search for a key x, we look at all the bits $BF[h_1(x)], \ldots, BF[h_t(x)]$. If any one of them is still 0, then we return "no." Had x been inserted into the BF, all of the bits would've been set to 1. If all of them are 1, then we return "probably yes."
    - Note: A search for x may find all the bits set to 1, even though x was never inserted.
      E.g. Suppose we use two hash functions, which maps x to bit positions 1 and 3, y to bit positions 1 and 2, and z to bit positions 2 and 3. If we insert y and z and search for x, the search alg will return "prob yes" even though x was never inserted. This is an example of a false positive.

- Insert Pseudo Code:

```
Insert(Fs, x)
    for i in range(1, t+1):
        BF[hi(x)] = 1
```

- Search Pseudo Code:

```
Search(Fs, x)
    for i in range(1, t+1):
        if (BF[hi(x)] == 0):
            return "no"
    return "prob yes"
```

- Assuming that we can evaluate each hash function in O(1) time, then the algs take $\Theta(t)$ time.
  Note: In typical uses, the number of hash functions used, t, is a small number, so the algs run in $\Theta(1)$ time.

- Probability of False Positives:
  - We can use multiple hash functions to reduce the probability of false positives.
  - The prob of a false positive depends on:
    1. The size of the filter, m.
    2. The number of things inserted, n.
    3. The number of hash functions, t.
  - If $\alpha = n/m$ goes up, then the number of collisions goes up and the number of false positives goes up.
  - If t is too low, there will be more collisions and false positives.

- If t is too high, then too many bits will be set to 1, which means that there will be more collisions and false positives.

- Assume that hash values are uniformly random and mutually independent. Suppose that we insert n keys into an empty BF and now lookup an absent key. To find the probability of returning true, we do:

  - We can model inserting n keys as randomly setting a bit to 1 nt times.

  - We can model searching for an absent key as (after) randomly picking t bits to read, what is the prob of getting all 1's?

- The probability that $h_i(k)$ hashes to j is $\frac{1}{m}$, so therefore, the prob that $BF[j]=0$ after one hash function is applied to one key is $1-1/m$.

- The prob that $BF[j]=0$ after t hash functions are applied to 1 key is $\left(1-\frac{1}{m}\right)^t$.

- The prob that $BF[j]=0$ after t hash functions are applied to n keys is $\left(1-\frac{1}{m}\right)^{nt}$.

- $$\begin{aligned} Pr(BF[i]=1) &= 1 - Pr(BF[i]=0) \\ &= 1-\left(1-\frac{1}{m}\right)^{nt} \\ &= 1-\left(1+\frac{(-nt)/m}{nt}\right)^{nt} \\ &\approx 1-e^{-nt/m} \\ &= 1-e^{-\alpha t} \end{aligned}$$

- To model search(Fs, k), we randomly check t bits in BF. The prob of getting all 1's is $(1-e^{-\alpha t})^t$.
  I.e. $Pr(\text{False Positive}) = (1-e^{-\alpha t})^t$.
  Note: We can minimize the prob of false positives based on the number of hash functions, t.

- The value of t that minimizes Pr(False Positive) is $t = \frac{\ln 2}{\alpha}$.

Proof:

$$P(t) = (1 - e^{-\alpha t})^t$$

$$\frac{dP(t)}{dt} = (1 - e^{-\alpha t})^t \left[\ln(1 - e^{-\alpha t})^t\right]'$$

$$= (1 - e^{-\alpha t})^t \left[t \ln(1 - e^{-\alpha t})\right]'$$

$$= (1 - e^{-\alpha t})^t \left[\ln(1 - e^{-\alpha t}) + \frac{\alpha t e^{-\alpha t}}{1 - e^{-\alpha t}}\right]$$

We can set $\frac{dP(t)}{dt}$ to 0 and solve for t. $t = 0$ or $\ln 2/\alpha$, and because we need a positive number of hash functions, $t = 0$ is not feasible

$\therefore t = \frac{\ln 2}{\alpha}$

- If we plug $t = \ln 2/\alpha$ back into the equation $(1 - e^{-\alpha t})^t$, we get:

$$\left(1 - e^{-\alpha\left(\frac{\ln 2}{\alpha}\right)}\right)^{\frac{\ln 2}{\alpha}}$$

$$= \left(1 - 2^{-1}\right)^{\frac{\ln 2}{\alpha}}$$

$$= \left(\frac{1}{2^{\ln 2}}\right)^{1/\alpha}$$

$$\approx (0.6185)^{1/2}$$

- E.g. 1 Given that $1/\alpha = 32$, what is the optimal t?

Soln:

$$t = \frac{\ln 2}{\alpha}$$
$$= (\ln 2)(32)$$
$$\approx 22$$

Plug t and $1/\alpha$ into the eqn $(1-e^{-\alpha t})^t$.

$$(1-e^{-\frac{22}{32}})^{22} \approx 2.1 \times 10^{-7}$$

- E.g. 2 Suppose that we want a prob of 10^{-7}. Find α and t.

Soln:

$$(0.6185)^{1/\alpha} = 10^{-7}$$
$$\frac{1}{\alpha} \approx 33.55$$
$$\alpha \approx \frac{1}{33.55}$$
$$t = \frac{\ln 2}{\alpha}$$
$$\approx 23$$

- We usually don't implement a delete function cause it will cause false negatives. However, in the case that we really need it, we can increment a counter at each location in BF. To delete, we decrement counters. However, BF is no longer a bit array.