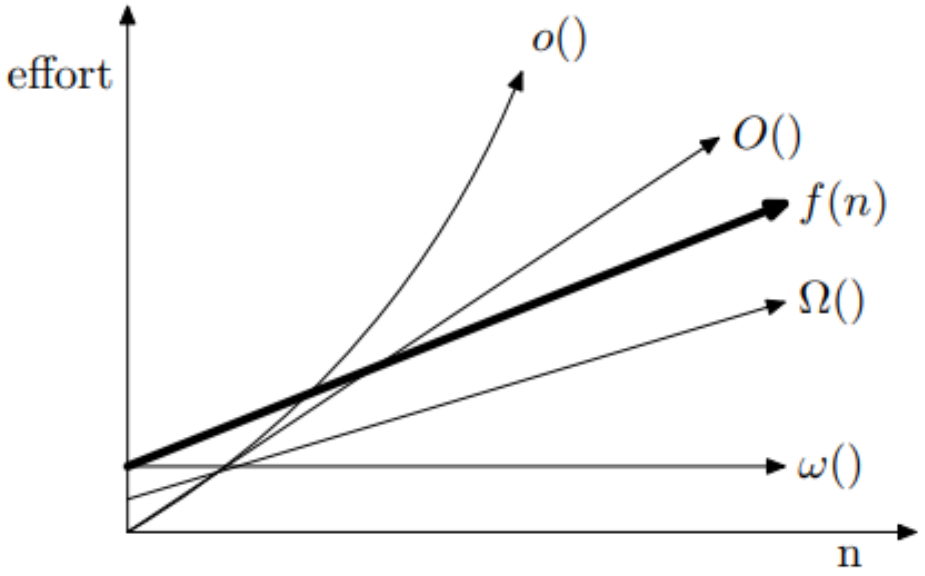
Pre-Final

335 Fall 2022

Reminder

$O(1) < O(\log(x)) < O(x) < O(x\log(x)) < O(x^2) < O(2^x) < O(x!) < O(x^x)$



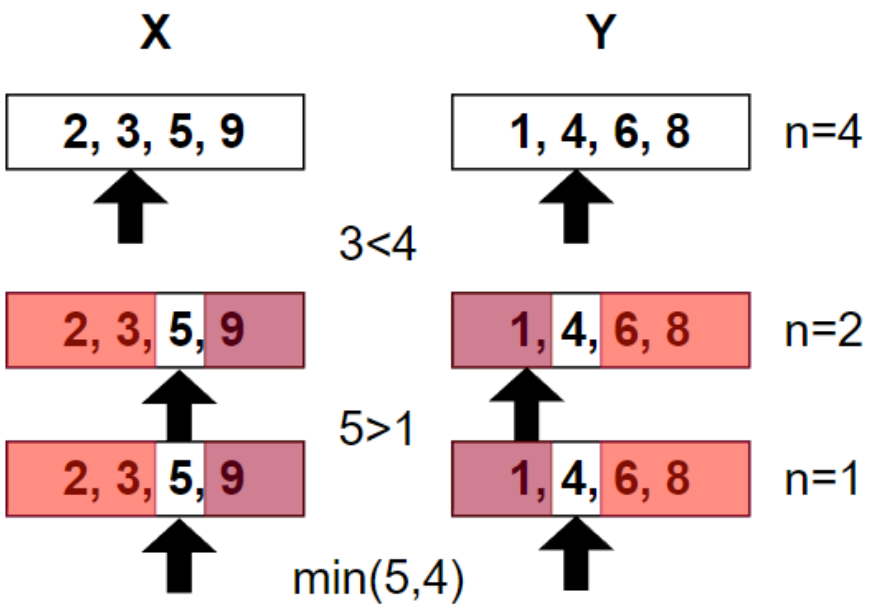
$T(n) = aT(n/b)+f(n)$

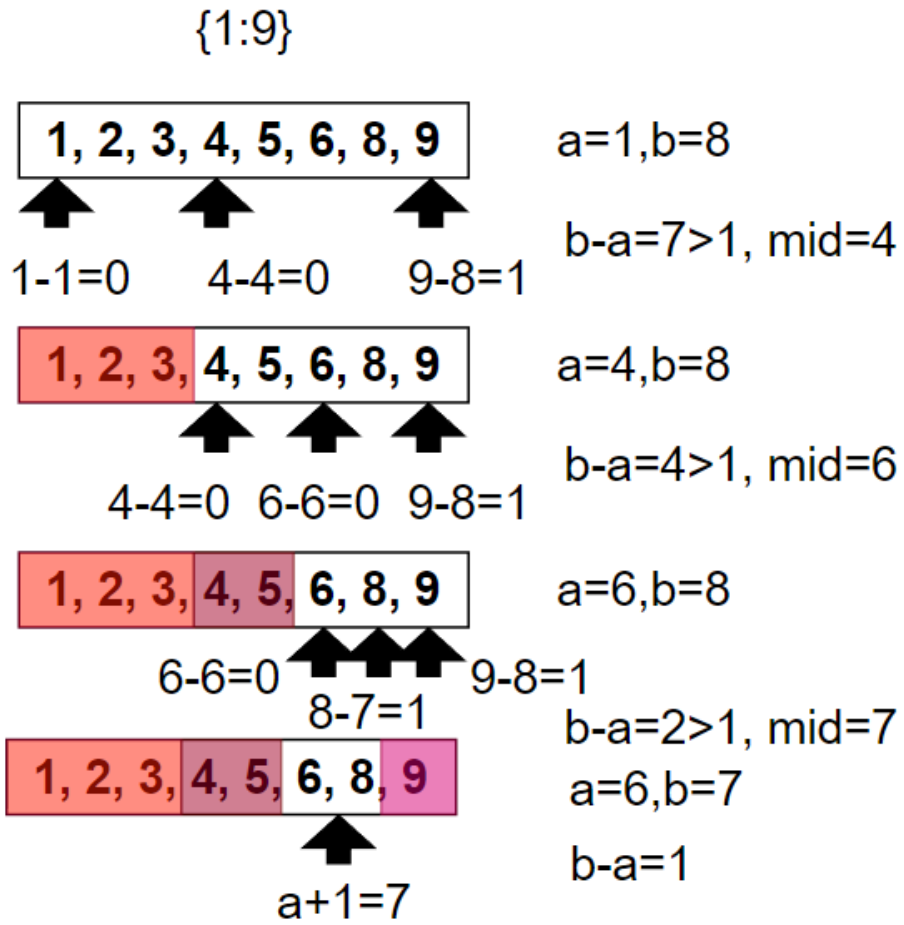
- Substitution Method
- Recurrence Tree
- Master Theorem

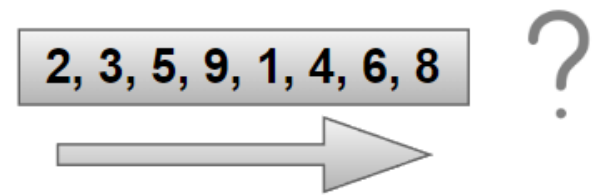
Order Statistics

- Give an O(logn) time algorithm to find the median of all 2n elements in sorted arrays X and Y with sizes n. Apply the algorithm to the arrays X:[2,3,5,9] and Y:[1,4,6,8].

- Give an O(logn) time algorithm to find the missing element from a sorted array of length n − 1 containing all but one element of {1, . . . , n}. Provide the recurrence relationships. Repeat for the unsorted array. Apply both on joining of two sequences X, and Y (from the previous example) with the missing element 7.







You can only traverse

Heapsort

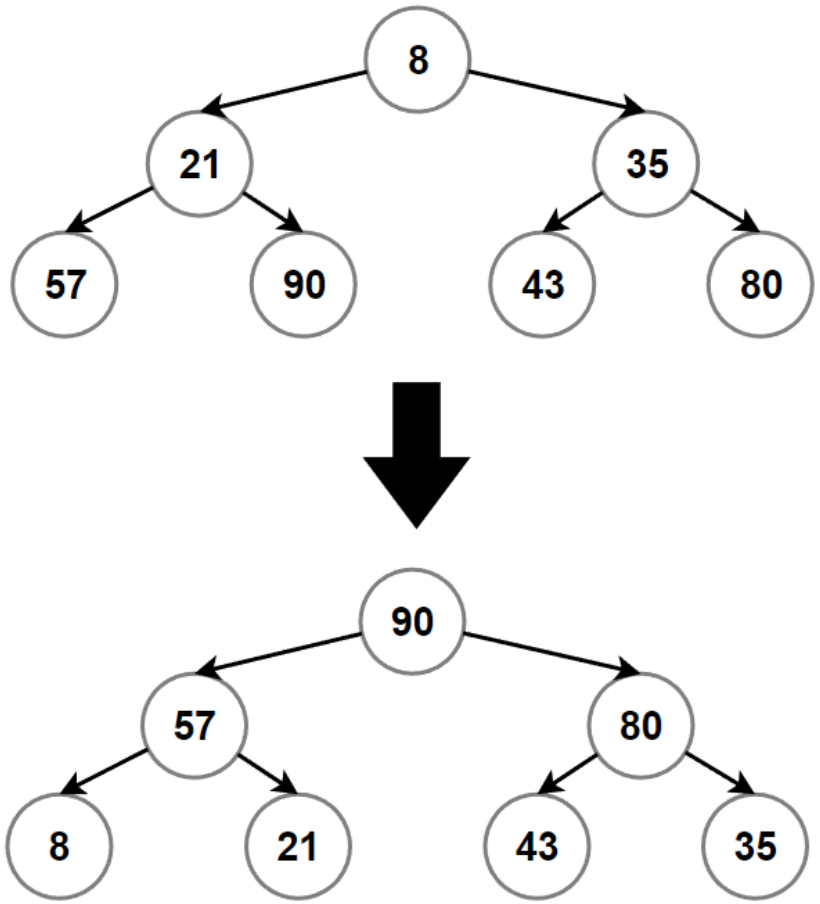
- Apply Heap-Sort to the sequence [8, 21, 35, 57, 90, 43, 80]
- What is the running time of HEAPSORT on an array A of length n that is already sorted in increasing order? What about decreasing order?

```
MAX-HEAPIFY(A, i)
1  l ← LEFT(i)
2  r ← RIGHT(i) ; largest ← i
3  if l ≤ heap-size[A] and A[l] > A[i]
4       then largest ← l
5       else largest ← i
6  if r ≤ heap-size[A] and A[r] > A[largest]
7       then largest ← r
8  if largest ≠ i
9       then exchange A[i] ↔ A[largest]
10           MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← floor(length[A]/2) downto 1
   do
3     MAX-HEAPIFY(A, i)
```

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← length[A] downto 2 do
3   exchange A[1] ↔ A[i]
4   heap-size[A] ← heap-size[A] – 1
5   MAX-HEAPIFY(A, 1)
```

8, 21, 35, 57, 90, 43, 80



This orderings will affect the Build-Max-Heap function which is upper bounded by O(n);

- If the is array sorted in descending order, the Max-Heapify call returns immediately: O(h)->O(1).
- If the is array sorted in ascending order, the Max-Heapify call always be the worst-case.

So for the both cases, the complexity will not change.

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) \qquad \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

Hash Tables

- Insert the keys [79, 69, 98, 72, 14, 50] into the Hash Table of size 13 where the main hash function is h'(k) = k mod 13 with the given c_1=,c_2=, and h_2(k) = 1 + (k mod 7).

  (see the Colab link: https://colab.research.google.com/drive/1HIigtVA9Ez_gbIDAZC7TsgzFtphSO0BY?usp=sharing)

- Consider an open-address hash table with uniform hashing. Give upper bounds on the expected number of probes in an unsuccessful search and on the expected number of probes in a successful search when the load factor is 3/4 and when it is 7/8.

  - ❖ Linear Probing -> $h(k, i) = (h'(k) + c_1 i) \bmod m$ -> Primary Clustering !
  - ❖ Quadratic Probing -> $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$ -> Secondary Clustering !
  - ❖ Double Hashing -> $h(k, i) = (h'(k) + i h_2(k)) \bmod m$ -> Double Function Selection !

Theorem 11.6

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

Theorem 11.8

Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha},$$

α=3/4

$$\frac{1}{1-\frac{3}{4}}$$

$$\frac{4}{3}\log(4)$$

α=7/8

$$\frac{1}{1-\frac{7}{8}}$$

$$\frac{8}{7}\log(8)$$

BST & Red-Black Trees

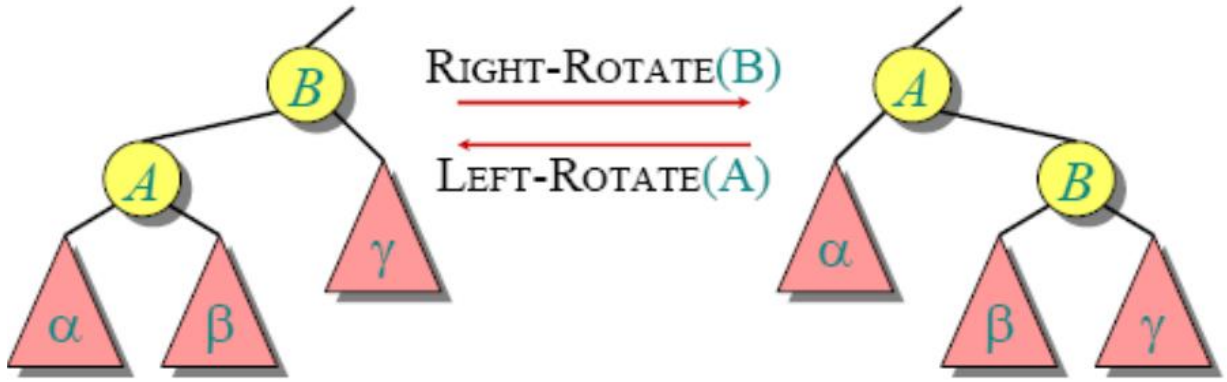
- Explain that no matter what node we start at in a height-h binary search tree, k successive calls to TREE-SUCCESSOR take O(h+k) time.
- Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. (Recall that every node is its own ancestor.)
- Show the red-black trees that result after successively inserting the keys [41, 38, 31, 12, 19, 8] into an initially empty red-black tree.
- Now show the red-black trees that result from the successive deletion of the keys in the order [8, 12, 19, 31, 38, 41].

- The worst case is where the asked node is the predecessor of the root. The number of steps to pass from the left subtree to the right subtree of the root is bounded by O(2h). The remaining is just the traversal which is bounded by O(3k). Therefore O(2h+3k) = O(h+k).

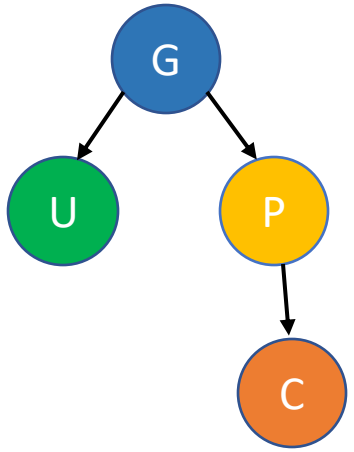
- Let y isn't the ancestor of x, then there should be a common ancestor of z. From the BST property, the relation between them will be $x<z<y$. Therefore, y won't become the successor of x.
- Let y's left child will be the ancestor of x (why it cannot be the right child, discuss). Therefore, there will be a lower ancestor z where $z<y$. Again, y won't become the successor of x.

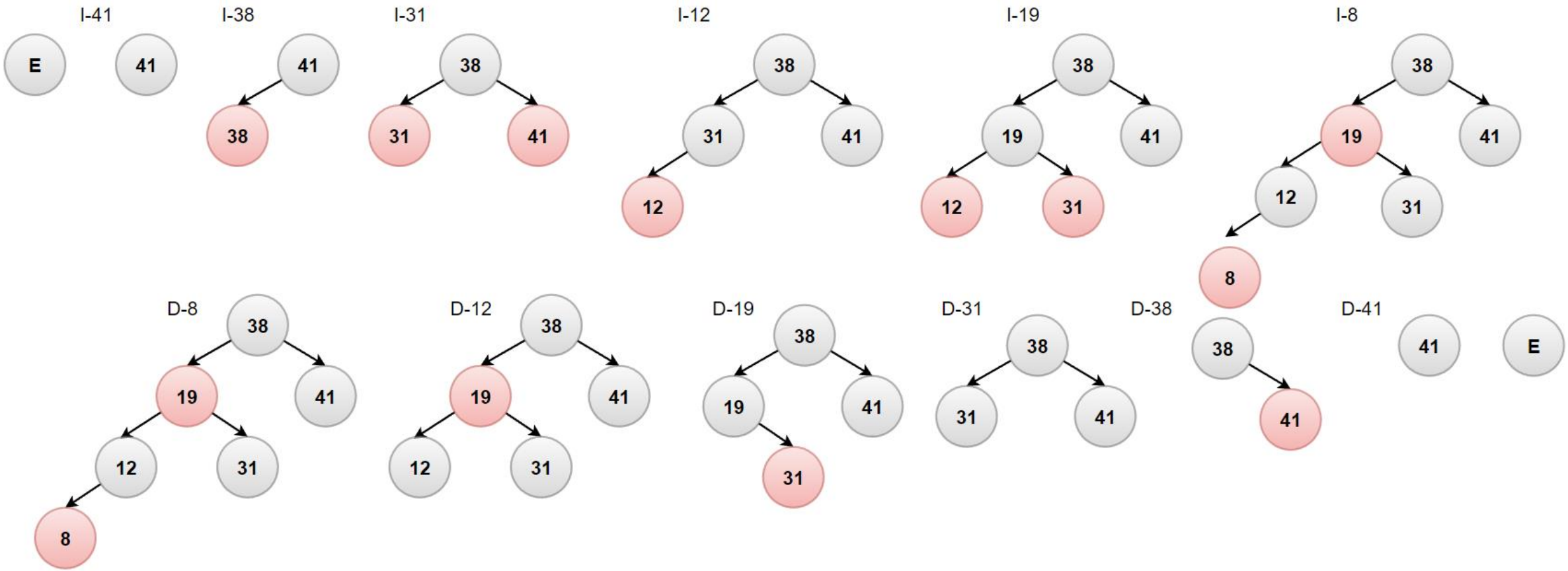
Reminder: Red-Black Trees

- The root and NILs are always **black**
- **Black condition**: every path from the root to a leaf node has the same number of **black** nodes
- **Red condition**: every **red** node has a **black** parent and **black** children



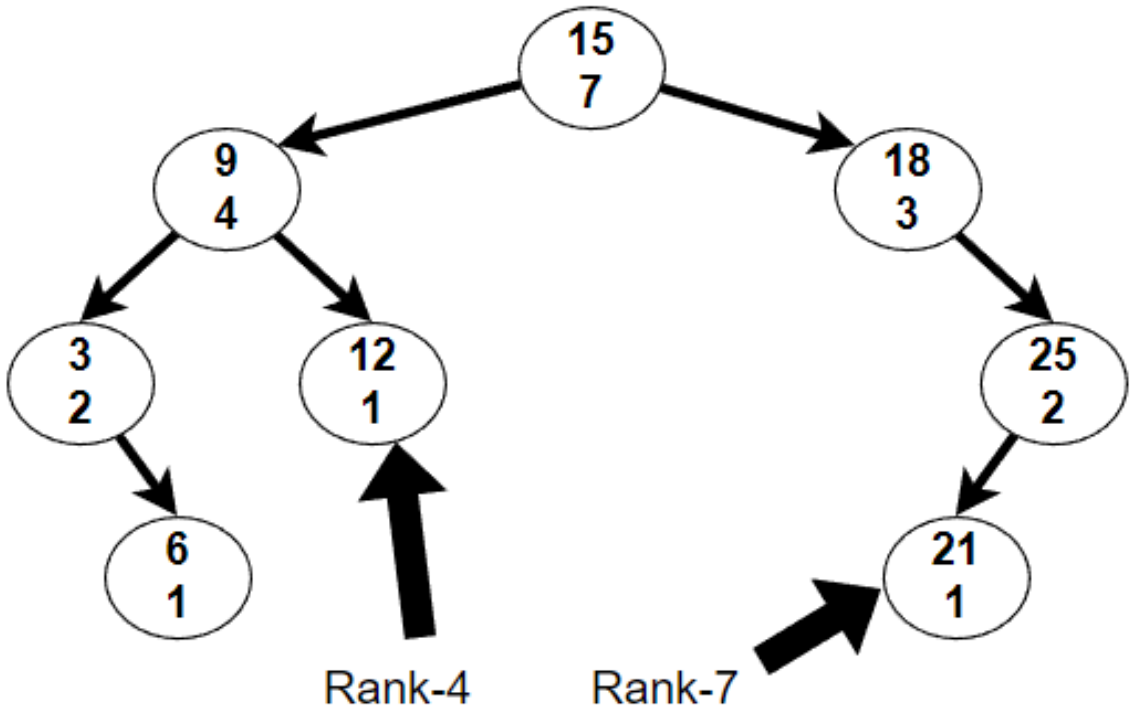
- Case 1: Uncle is **red** -> Recolor
- Case 2: Uncle is **black**:
  - Case 2-a: grandparent-parent-child forms a line -> Rotate & Recolor
  - Case 2-b: grandparent-parent-child forms a triangle-> Transform it to case 2-a




I-41
E
41
I-38
41
38
I-31
38
31
41
I-12
38
31
41
12
I-19
38
19
41
12
31
I-8
38
19
41
12
31
8
D-8
38
19
41
12
31
8
D-12
38
19
41
12
31
D-19
38
19
41
31
D-31
38
31
41
D-38
38
41
D-41
41
E

Augmenting Data Structures

- Find rank-4 value and of rank of value 21 from the next OST.
- Show, by adding pointers to the nodes, how to support each of the dynamic-set queries MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in O(1) worst-case time on an augmented OST (the asymptotic performance of other operations on OST should not be affected).
- Explain how would you implement a stack using queue(s) and vice-versa.



OS-SELECT(x, i)

```
1  r = x.left.size + 1
2  if i == r
3        return x
4  elseif i < r
5        return OS-SELECT(x.left, i)
6  else return OS-SELECT(x.right, i − r)
```

OS-RANK(T, x)

```
1  r = x.left.size + 1
2  y = x
3  while y ≠ T.root
4        if y == y.p.right
5              r = r + y.p.left.size + 1
6        y = y.p
7  return r
```

- Add predecessor and successor pointers to each node. Also, connect the leftmost and the rightmost leaf to make the traversal circular. Lastly, add an additional pointer to the root that points to either the leftmost or the rightmost leaf. (discuss how to maintain tree structure)

- We need two queues, and mark one of them as the 'current' and the other as 'other'. There is two options;

1. Keep the pop as it is and pop from the current. Whenever a push needed, transfer every element in current to the other, then push it to other one. Lastly, transfer every element in current from the other to the current.
2. Interchange pop and push from the previous explanation.

- Interchange 'queue' and 'stack' words at the previous explanation.

Amortized Analysis

- Let's revisit recitation from the related week.

Basic Operations on B-Trees (18.2-1)

- Show the results of inserting the keys {F; S; Q; K; C; L; H; T; V; W; M; R; N; P; A; B; X; Y; D; Z; E } in order into an empty B-tree with minimum degree 2. Draw only the configurations of the tree just before some node must split, and also draw the final configuration.

Min Keys = 1
Max Keys = 3

F S Q K C L H T V W M R N P A B X Y D Z Ɛ

[F]

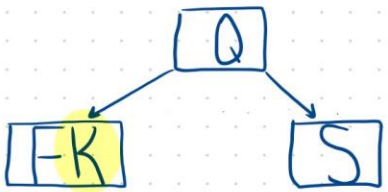
F S Q K C L H T V W M R N P A B X Y D Z Ɛ

[F S]

F S Q K C L H T V W M R N P A B X Y D Z Ɛ

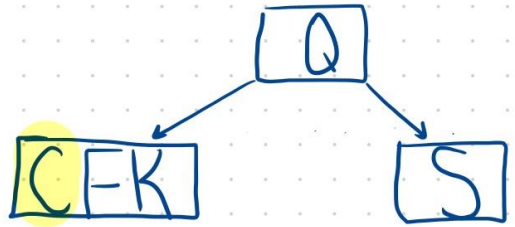
[F Q S]

F S Q K C L H T V W M R N P A B X Y D Z Ɛ

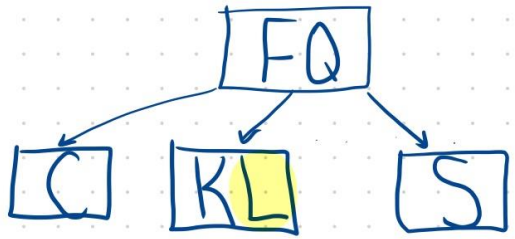


STEP 1: NEW LEVEL
STEP 2: INSERT KEY

F S Q K C L H T V W M R N P A B X Y D Z Ɛ

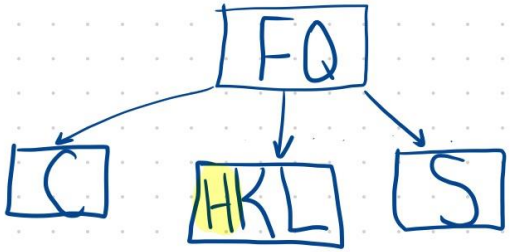


F S Q K C L H T V W M R N P A B X Y D Z Ɛ



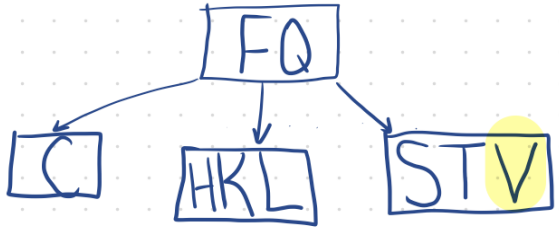
STEP 1: SPLIT NODE
STEP 2: INSERT KEY

F S Q K C L H T V W M R N P A B X Y D Z Ɛ

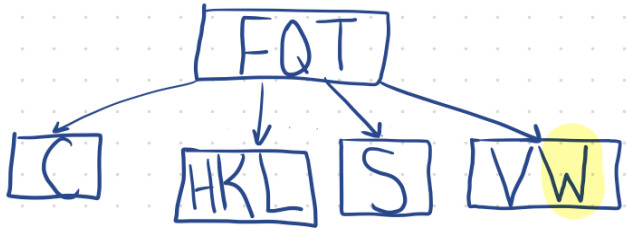


FSQKCLHTVWMRNPABXYDZE

FSQKCLHTVWMRNPABXYDZE

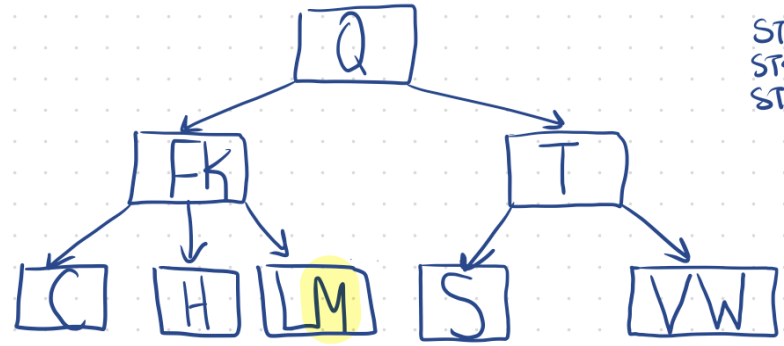


FSQKCLHTVWMRNPABXYDZE



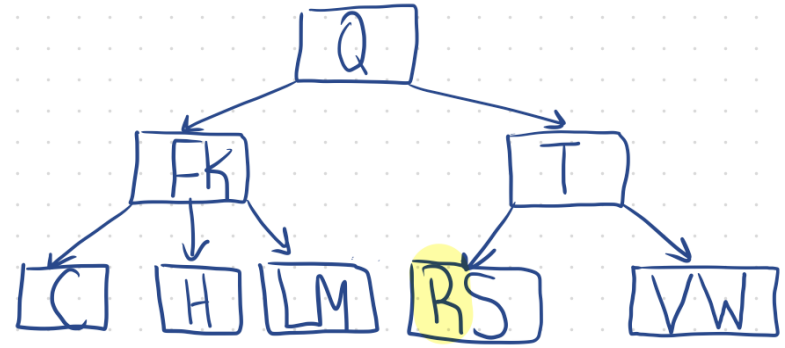
STEP 1: SPLIT NODE
STEP 2: INSERT KEY

FSQKCLHTVWMRNPABXYDZE

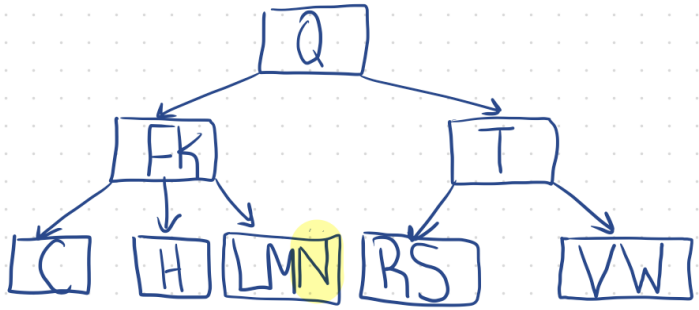


STEP 1: NEW LEVEL
STEP 2: SPLIT NODE
STEP 3: INSERT KEY

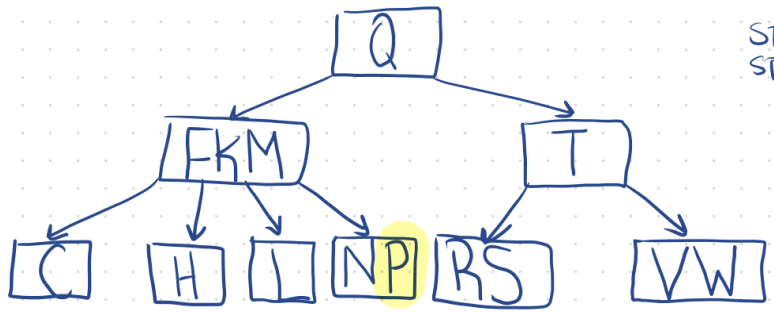
FSQKCLHTVWMRNPABXYDZE



F S Q K C L H T V W M R N P A B X Y D Z E

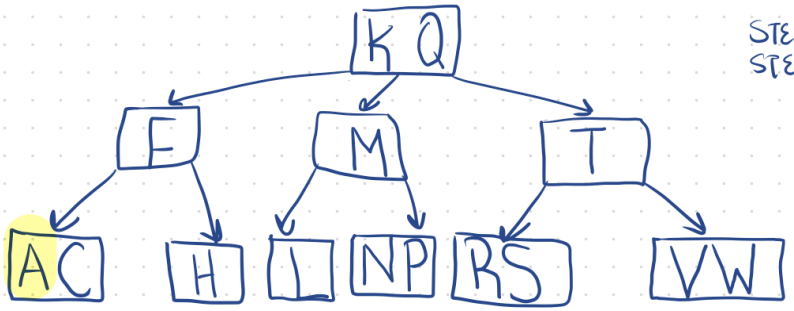


F S Q K C L H T V W M R N P A B X Y D Z E



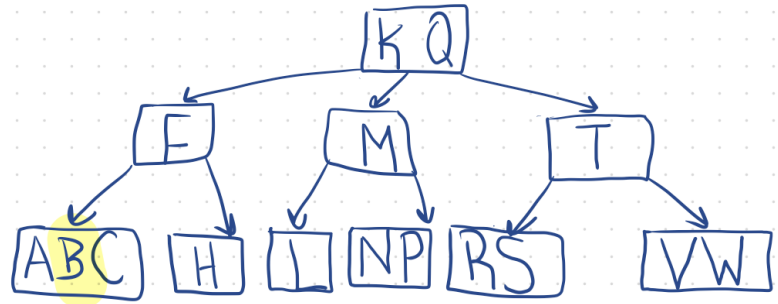
STEP1 : SPLIT NODE
STEP2 : INSERT KEY

F S Q K C L H T V W M R N P A B X Y D Z E

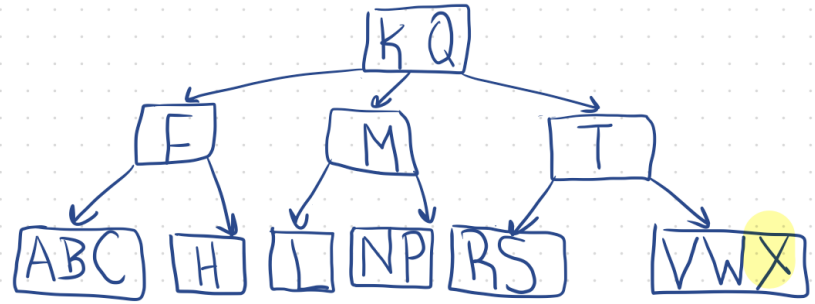


STEP1 : SPLIT NODE
STEP2 : INSERT KEY

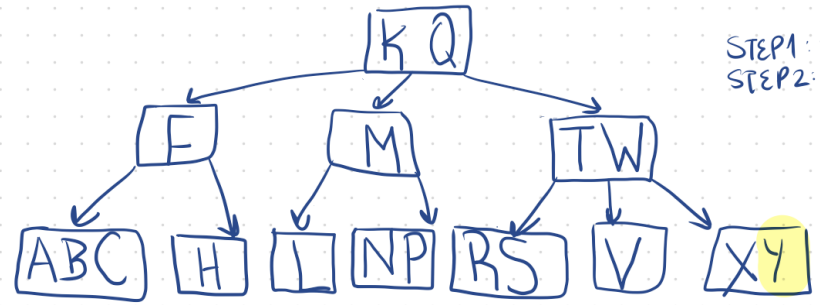
F S Q K C L H T V W M R N P A B X Y D Z E



F S Q K C L H T V W M R N P A B X Y D Z E

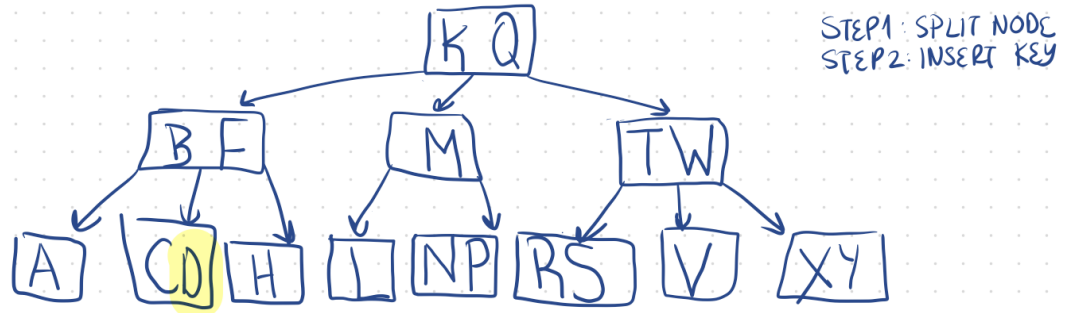


F S Q K C L H T V W M R N P A B X Y D Z E



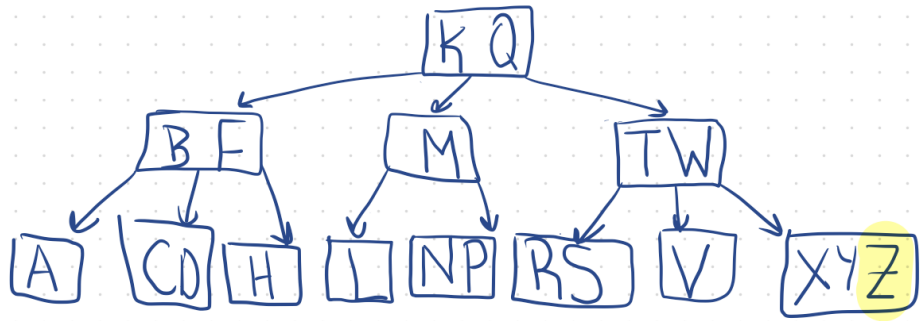
STEP1 : SPLIT NODE
STEP2 : INSERT KEY

FSQKCLHTVWMRNPABXYDZE

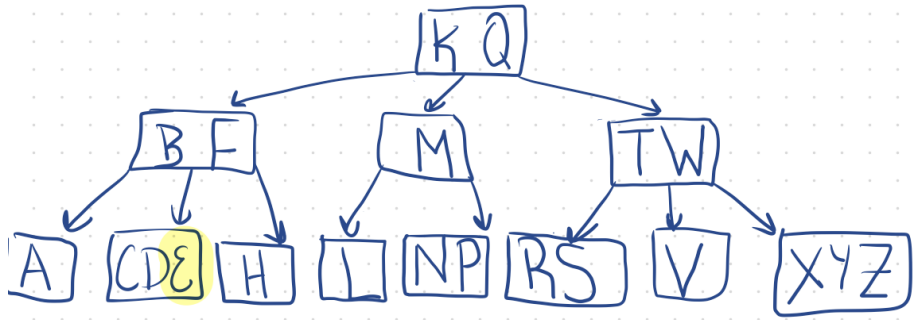


STEP1: SPLIT NODE
STEP2: INSERT KEY

FSQKCLHTVWMRNPABXYDZE

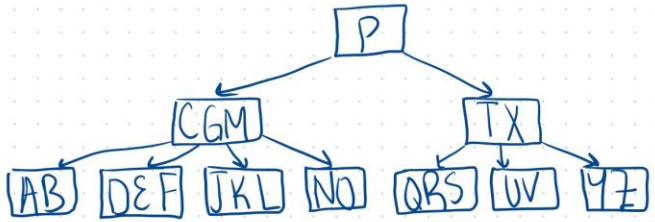


FSQKCLHTVWMRNPABXYDZE

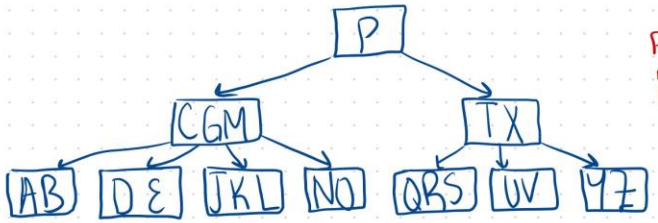


Deleting a Key From a B-Tree (18.3-1)

- Show the results of deleting C, P, and V, in order, from the tree of figure 18.8(f).

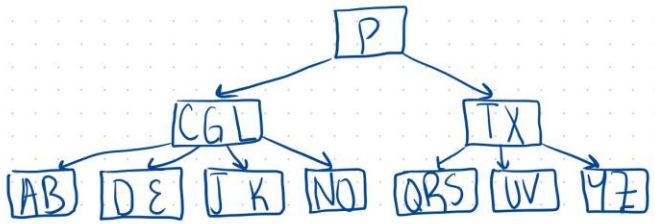


DELETE F (CASE 1):



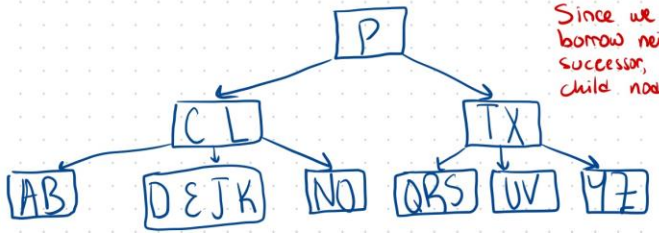
F is simply a leaf node and it can be removed.

DELETE M (CASE 2a):



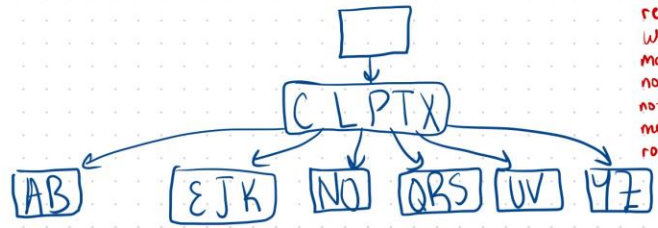
L, The predecessor of M is borrowed to maintain branching factor of CGM node

DELETE G (CASE 2c):



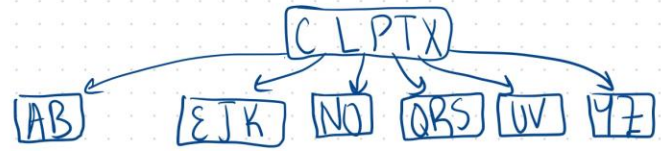
Since we cannot borrow neither predecessor nor successor, we merge the two child nodes of the G key

DELETE D (CASE 3b):



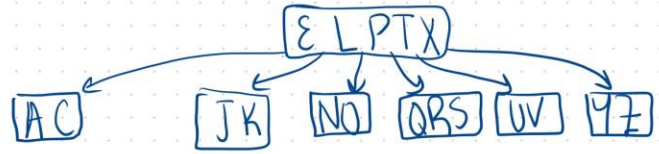
We hit node CL & realize it only has 2 keys We want to proactively maintain the degree of the node. We check its sibling and notice it too has few keys. We merge the two siblings & the root (as the median)

D is then deleted as it is found to be a leaf node.



The tree shrinks in depth.

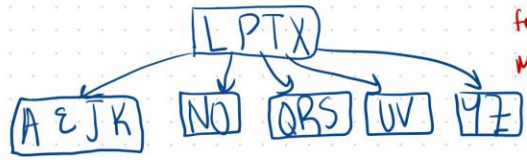
DELETE B (CASE 3A):



Check B's sibling to transfer a key to maintain branching factor. (Left Rotate)

Then delete B.

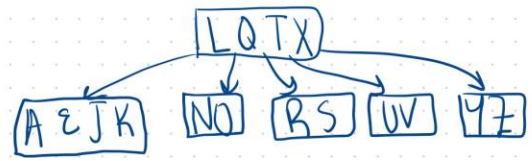
DELETE C (CASE 3b):



C's sibling has too few keys to rotate.

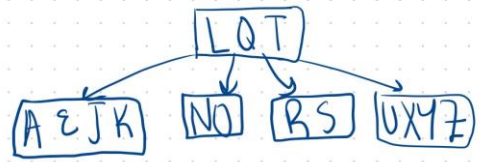
Merge & bring down E as median

DELETE P (CASE 2a):



Bring Q, the successor of P into P's place.

DELETE V (CASE 3b):



To maintain the branching factor of UV, we can either borrow from a sibling, or merge with a sibling with the common parent as the median. In this case, neither sibling has enough keys to lend one, so we merge.

Basic Operations on B-Trees (18.2-4)

- Suppose that we insert the keys {1, 2, …, n} into an empty B-tree with minimum degree 2. How many nodes does the final B-tree have?

Solution (18.2-4)

- The final tree can have as many as $n-1$ nodes. Unless $n=1$ there cannot ever be n nodes since we only ever insert a key into a non-empty node, so there will always be at least one node with 2 keys.
- Next observe that we will never have more than one key in a node which is not a right spine of our B-tree. This is because every key we insert is larger than all keys stored in the tree, so it will be inserted into the right spine of the tree. Nodes not in the right spine are a result of splits, and since $t=2$, every split results in child nodes with one key each. The fewest possible number of nodes occurs when every node in the right spine has 3 keys. In this case, $n=2h+2^{h+1}-1$where h is the height of the B-tree, and the number of nodes is $2^{h+1}-1$. Asymptotically these are the same, so the number of nodes is $\Theta(n)$.