DSA

03 August 2023 07:43 PM

$f(x) = O(g(n))$ if and only if there exists a M

$f(x) \leq M\, g(n)$ $(M > 0) \in \mathbb{R}$ $(n \geq n_0) \in \mathbb{N}$

small $o(g(n)) = f(n)$

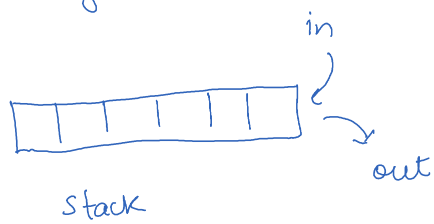
means $f(n) < c_1 g(n)$ strictly less

STACKS

- → Accessing $O(N)$
- → searching $O(N)$
- → insertion $O(1)$
- → deletion $O(1)$

Linear DS

LIFO



QUEUES

- → Accessing $O(N)$
- → searching $O(N)$
- → insertion $O(1)$
- → deletion $O(1)$

Linear DS

FIFO

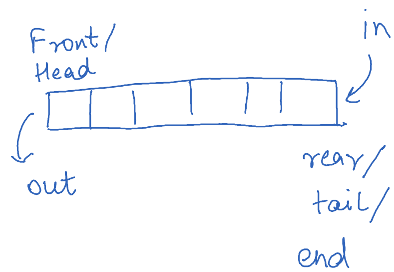
types:-

Simple

Circular

Double ended

Priority



DEQUE

- → double ended Queue
- → insertion & deletion at both ends
- → supports stacks and queues.

LINKED LIST

- → searching $O(N)$
- → Accessing $O(N)$
- → deletion $O(1)$
- → insertion $O(1)$

types →

singly LL

doubly LL

circular LL

doubly circular LL

CIRCULAR QUEUE

F = front E = end

$F \% \text{size} = (R+1) \% \text{size}$ then full.

STL QUEUES

`queue <int> x;`

- → enqueue (int x) insert
- → dequeue () deletes
- → front () returns front element
- → rear () returns rear element
- → size () returns size
- → push (int x) pushes element x
- → pop ()
- → empty () boolean

STACK STL

`stack <int> s;`

- → push (int x) pushes x
- → pop () pops
- → empty () returns boolean.
- → size () returns size
- → top () returns top element

HASHES

- → maps are example.
- → components: key, hash func^n, hash table

$$\underbrace{h}_{\text{func}^n}(\underbrace{x}_{\text{key}}) = \underbrace{y}_{\text{index of hash table}}$$

Types of Hash functions:

1) Division method: $h(k) = k \% M$

   k = input key
   M = prime no., size of table

2) Mid Square Method: extract middle r digits from k^2

3) Digit folding Method:

   divide k into parts $k_1\ k_2\ k_3 \ldots$ each having equal number of digits
   ([illegible] have equal digits)

divide k into parts $k_1\ k_2\ k_3 \dots$ each having equal number of digits
(last one may or may not have equal digits)

$h(k) = k_1 + k_2 + k_3 \dots$ ignore the last carry if it exists.

4) Multiplication Method:

$0 < A < 1$ constant

M size of hash table

$$h(k) = \text{floor}\left[M \ (kA \ \% \ 1) \right]$$

Collission handling:

- Open Hashing (Separate chaining)
- Open Addressing
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

1) Separate Chaining:

Each cell of the hash table point to a Linked List of records that have the same hash func^n value.

2) Linear Probing:

→ if location empty then insert.

→ else we search for the sequentially next free location

3) Quadratic Probing

→ if $h(k)$ occupied

check $(h(k) + 1^2) \ \% \ M$

then check $(h(k) + 2^2) \ \% \ M$

⋮

till you find a free location.

4) Double Hashing

→ $h_1(k)$ initial hash func^n. If location is empty then insert

→ else we use $h_2(k)$

$[h_1(k) + i\, h_2(k)] \ \% \ M$

→ else we use $h_2(k)$

$$H(k) = \left[h_1(k) + i\, h_2(k)\right] \% M$$

Applications:

→ Data Integrity (encryption)

→ Password verification.

→ Data Storage.

→ Mapping.

Load factor:

$$\alpha = \frac{\text{no. of items}}{\text{size of table}}$$

$\alpha \leq 0.75$ always.

α tells the load of each entry in the hash table.

If $\alpha > 0.75$ then increase the size of table.

M = existing size.

new size $\geq 2 * M$

↳ should be prime.

TREES

Types of Binary Trees

1) full BT ⇒ every node has 0 or 2 children.
2) Complete BT ⇒ all the (l-1) level nodes must have 2 children and the last level must be filled from left to right.
3) Degenerate BT ⇒ formation of a linked list.
4) Perfect BT ⇒ All leafs are at the same level.

Types of trees

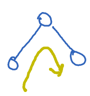
1) Binary tree
2) Ternary tree
3) m-array tree.

Types of trees.

1) Binary Search Tree
2) AVL tree ($|h_L - h_R| \le 1$) → height balanced tree
3) RB tree
4) B tree
5) B^+ tree
6) Splay tree.

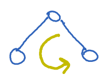
Tree transversal Algorithm (DFS):

1) inorder



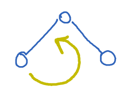
left root right

2) Pre order



root left right

3) Post order



left right root

Properties

\# nodes at level l $\in [1, 2^l]$ root is level 0

\# nodes in BT of height h = $2^{h+1} - 1$

min levels with n nodes = floor($\log_2(n)$)

L leaves has atleast ceil($\log_2(L)$) levels

min height with n nodes = $\log_2(n+1) - 1$

\# leaves = # internal nodes + 1

\# edges = # nodes − 1

Complexities (BST)

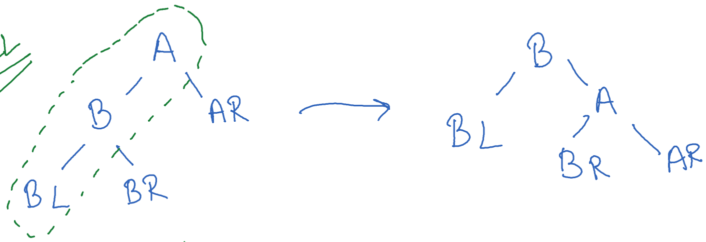
1) insertion $O(h)$ h = height
2) deletion $O(h)$

Reconstruction.

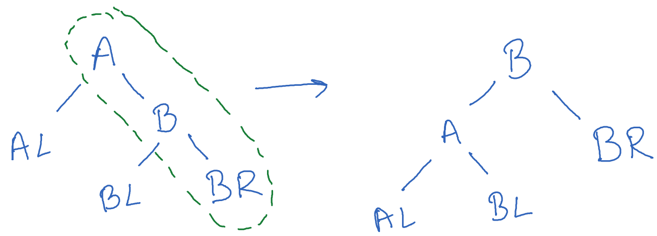
inorder transversal is required to reconstruct the tree always

INSERTION ROTATIONS (AVL TREE)

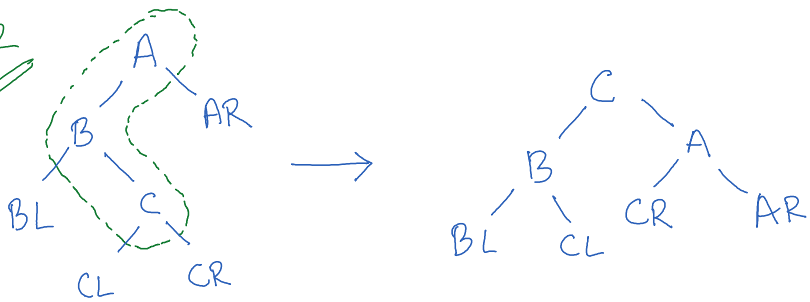
LL



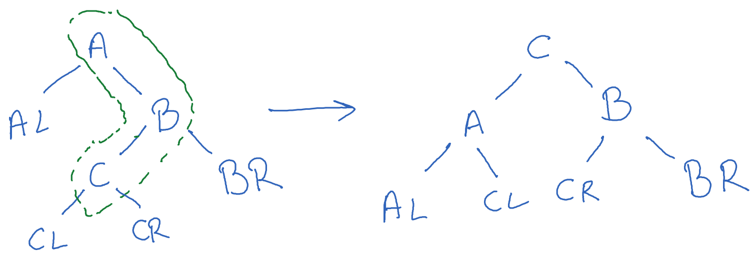
RR



LR

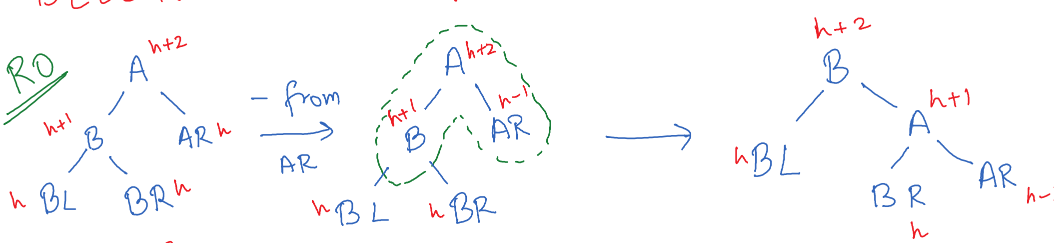


RL

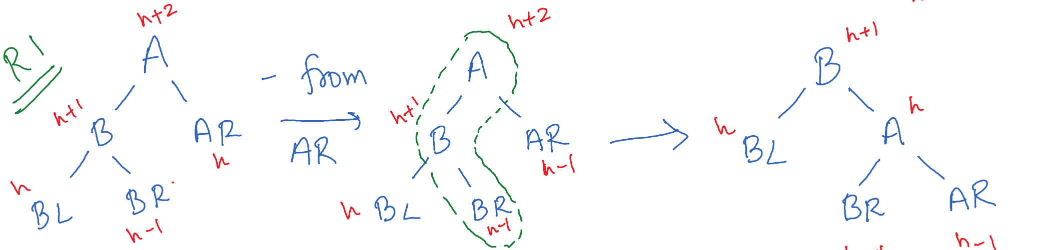


DELETION ROTATIONS (AVL TREE)

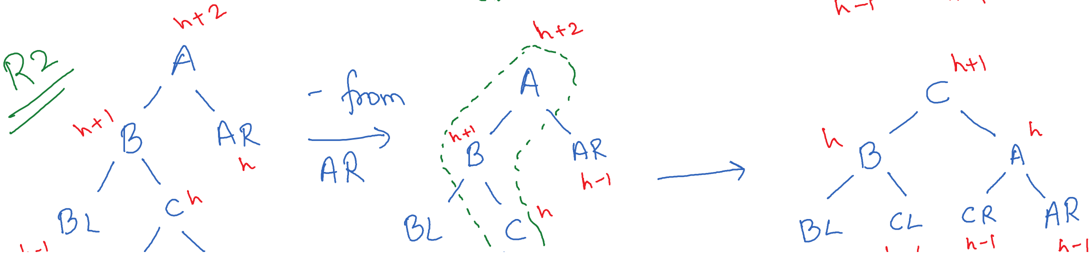
R0

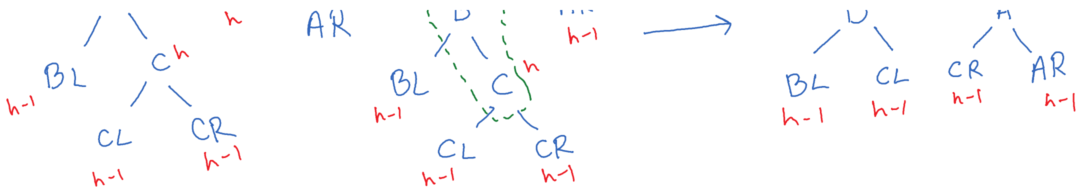


R1



R2





B - TREE

- m - array tree
- reduce height
- store data on hard drive
- leaves are at same level
- all operations in $O(\log n)$

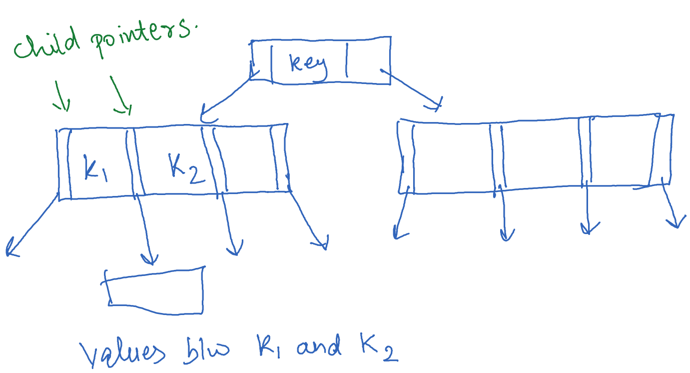
Props:

- The tree is defined around a constant t
- Every node (except root) must contain atleast $t-1$ Keys
- Every node must contain atmost $2t-1$ Keys.
- # children = # keys + 1

$\therefore$ min degree = $(t-1)+1 = t$ degree = no. of children.

max degree = $(2t-1)+1 = 2t$

- the Keys are in sorted ascending order.
- root has a minimum of 1 key



- if n = total Keys in B-Tree

$h_{min} = \log_{2t}(n+1) - 1$ $h_{max} = \log_t\left(\frac{n+1}{2}\right)$

- Number of nodes and number of Keys are different.

TIME COMPLEXITIES:

Time Complexities:

	Binary Tree	BST	B-Tree	AVL Tree
insertion	$O(n)$	$O(h)$	$O(\log n)$	$O(\log_2 n)$
deletion	$O(n)$	$O(h)$	$O(\log n)$	$O(\log_2 n)$
Searching	$O(n)$	$O(h)$	$O(\log n)$	$O(\log_2 n)$

Histogram Area Using Stacks

Algorithm:

- Initialise a stack **S**.
- Push the first index of **A[]** into the stack.
- Traverse through the array **A[]** and compare the height of **A[i]** with the height at the top of the stack.
- If the height is:
  - Greater than **A[S.top()]**, push it into the stack.
  - Less than **A[S.top()]**, keep popping the elements until **A[i] >= A[S.top()]**.
- Keep maximizing the area while popping the elements from the stack.
- Push the index **i** for each element.
- Return the maximum element.

```
int largestRectangleArea(vector < int > & heights) {
  stack < int > stk;
  stk.push(-1);
  int max_area = 0;
  for (size_t i = 0; i < heights.size(); i++) {
    while (stk.top() != -1 and heights[stk.top()] >= heights[i]) {
      int current_height = heights[stk.top()];
      stk.pop();
      int current_width = i - stk.top() - 1;
      max_area = max(max_area, current_height * current_width);
    }
    stk.push(i);
  }
  while (stk.top() != -1) {
    int current_height = heights[stk.top()];
    stk.pop();
    int current_width = heights.size() - stk.top() - 1;
    max_area = max(max_area, current_height * current_width);
  }
  return max_area;
}
```

B-tree is faster than AVL tree

Both have logarithmic time complexity but the base is diff

in AVL tree $O(\log_2 n)$ in B tree $O(\log_t n)$

generally $t = 50$ to 2000

Insertion in B-tree

Rules:

→ insertion only occurs at leaf nodes

→ All leaf nodes must be at the same level

→ Whenever we encounter a fully filled node when travelling from root to leaf, we split the node around the middle value.

Time complexity

→ $O(h)$ h = height of tree.

→ The tree grows upwards (AVL tree grows downwards)

Variants of B-trees:

B^+-tree

→ data stored only at leaf nodes

→ all leaf nodes are linked to each other for navigation

B^*-tree

→ B trees require node to be atleast 1/2 full

→ B^* trees require node to be atleast 2/3 full

2-3-4 tree

→ $t = 2$

→ simplest B tree

Uses of B-tree.

→ Multi level indexing

→ Storing data on hard drives

Deletion

Case 1 : Delete from leaf

→ can delete directly

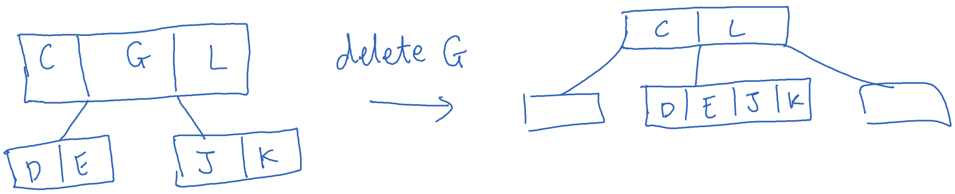
Case 2 a,b : Delete from internal nodes

Check whether any of its left or right child has at least t keys.

if yes, then bring that key in place of the key that is to be deleted.

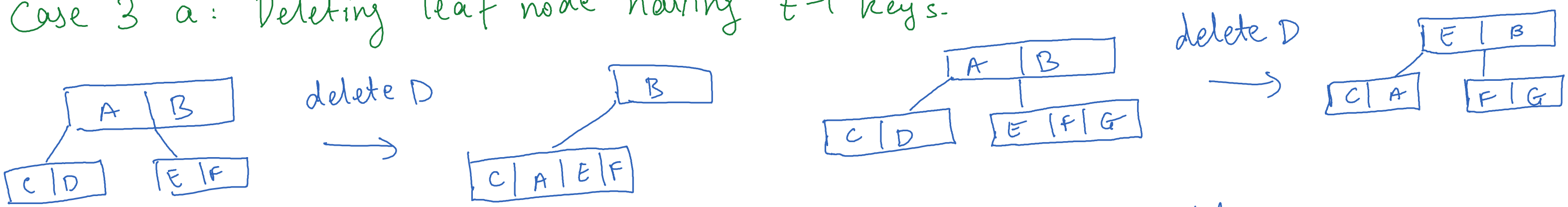
$t = 2$

Case 2c : If none of the childs have atleast t keys.



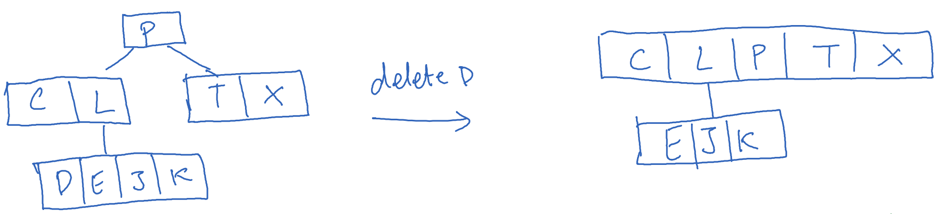
merge the left and right child & then delete the key.

Case 3 a: Deleting leaf node having t-1 keys.



Bring one of the parent keys down & merge the children. then delete.

Case 3b: Deleting leaf node, when its parents are underfull.



Rules: → if a node is full, split it.
→ if a node is underfull, merge it

} While traversing from root to leaf.

→ If you cannot borrow, merge.

in 2 a,b we look at child
in 3 we look if nodes are underfull.
in 1 we directly delete.

Order vs Degree.

B-tree degree = t   min children = t   Max children = 2t.

B-tree order = m   min childre = $\lceil \frac{M}{2} \rceil$ Max children = m.

HEAPS

→ Almost complete B tree (filled from left to right)

→ Types: Max Heap: Parent node larger than both of its children

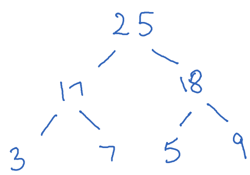
→ Almost complete B tree (filled from left to right)

→ Types: Max Heap : Parent node larger than both of its children

Min Heap: Parent node smaller than both of its children

Binomial Heap

Fibonacci Heap



Heaps represented as array :

1	2	3	4	5	6	7
25	17	18	3	7	5	9

first parent, then left & right child. Then left childs children

i) if node = i

parent node = $\text{floor}\left(\frac{i}{2}\right)$

ii) if node = i

left child = $2i$

right child = $2i+1$

Algorithm for building Heaps:

i) Heapify (heap A, index i)

ii) Build Max-Heap (heap A, length n)

Heapify Algorithm

```
heapify ( int arr[ ], int N , int i)
{
        Largest = i
          l = 2i
          r = 2i+1
        if ( N > l && arr[l] > arr[largest])
               largest = l
        if ( N > r && arr[r] > arr[largest] )
               largest = r
```

```
        if ( largest ! = i )
        {
                swap ( arr [ i ], arr [largest] )

                heapify ( arr[ ], N, largest )

         }
}
```

heapify time complexity is $O(h)$ or $O(\log n)$ $\quad h = \log n$

Build Max-Heap Algorithm

* first build a almost complete BT arr[ ] with elements inserted

from $\lceil \frac{n}{2} \rceil$ to n are leaf nodes.

```
from i = ⌈n/2⌉ to 1

        Heapify (arr, N, i)
```

Build Max-Heap time complexity = $O(n)$

although it is called $\frac{n}{2}$ times with heapify taking $O(n)$, so technically it should be $O(hn)$. But this is not asymptotically tight.

Heap Sort

→ replace the root with the last element in array

→ again build Max-Heap, but now with N-1 array

→ continue this till you only have one element left in the array

```
for i = n to 1
{
      Swap arr[1] with arr[i]
      Heapify ( arr[ ], i-1, 1 )
}
```

heap size ($i-1$), index (1)

* assumption : indexing starts at $1 \rightarrow n$

time complexity = $\underbrace{O(n\log n)}_{\text{sort}} + \underbrace{O(\log n)}_{\text{build}} = O(n\log n)$

Adv

→ Doesn't require extra space

→ Used as a priority queue.

Infix to Postfix expressions.

→ put the variables in the output stack directly

→ if an opr has higher or equal preference than the last opr in opr stack, then add that opr to the stack.

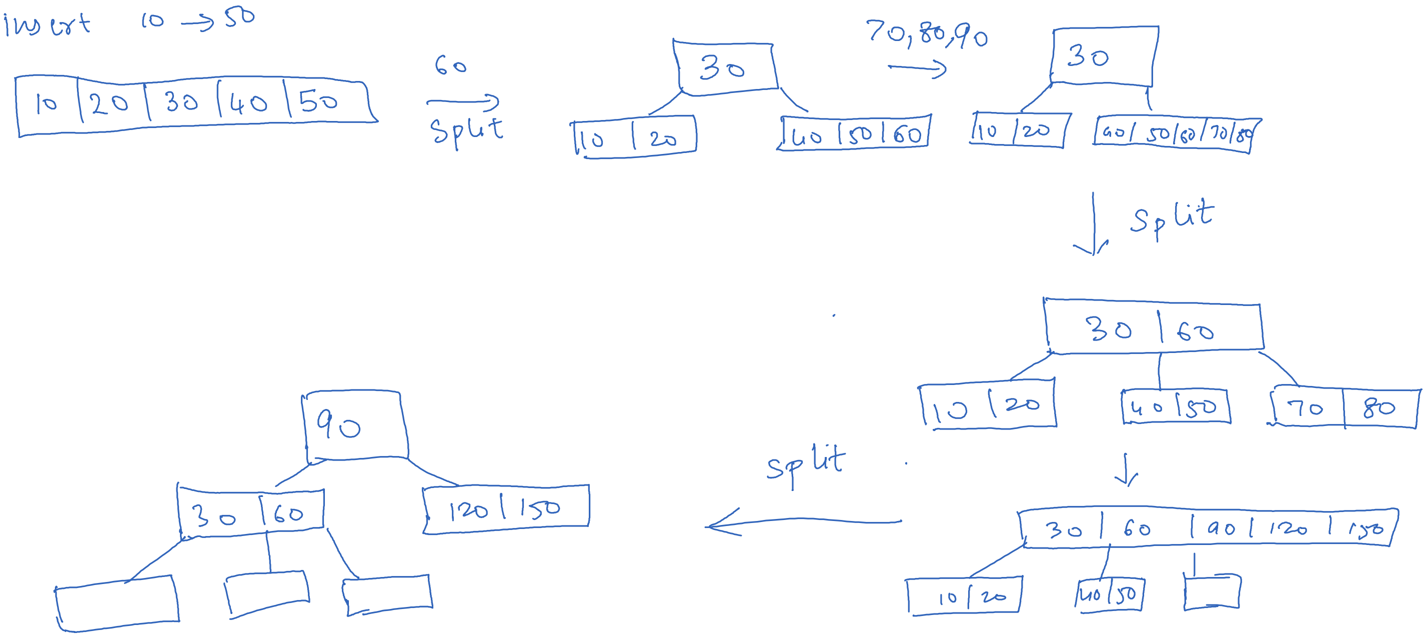
→ if an opr doesn't have higher preference than the last opr in opr stack, then remove all opr from opr stack till the last opr in opr stack has strictly less preference than the encountered operator.

Insertion in B tree

insert 10, 20, 30, 40, 50, 60, 70, 80, 90 $t = 3$

Max = $2t - 1 = 5$

insert 10 → 50



Deletion in B trees Case 3

We have to delete key x

We find x in node y

if y has $t-1$ keys then we perform this case.

Case 3a: if one of the siblings of y has t keys.
perform something like 2a, 2b to make y have t keys
Then delete x

Case 3b: if both siblings of y have $t-1$ keys.
perform Something like 2c to make y have more than $t-1$ keys.
Then delete x.