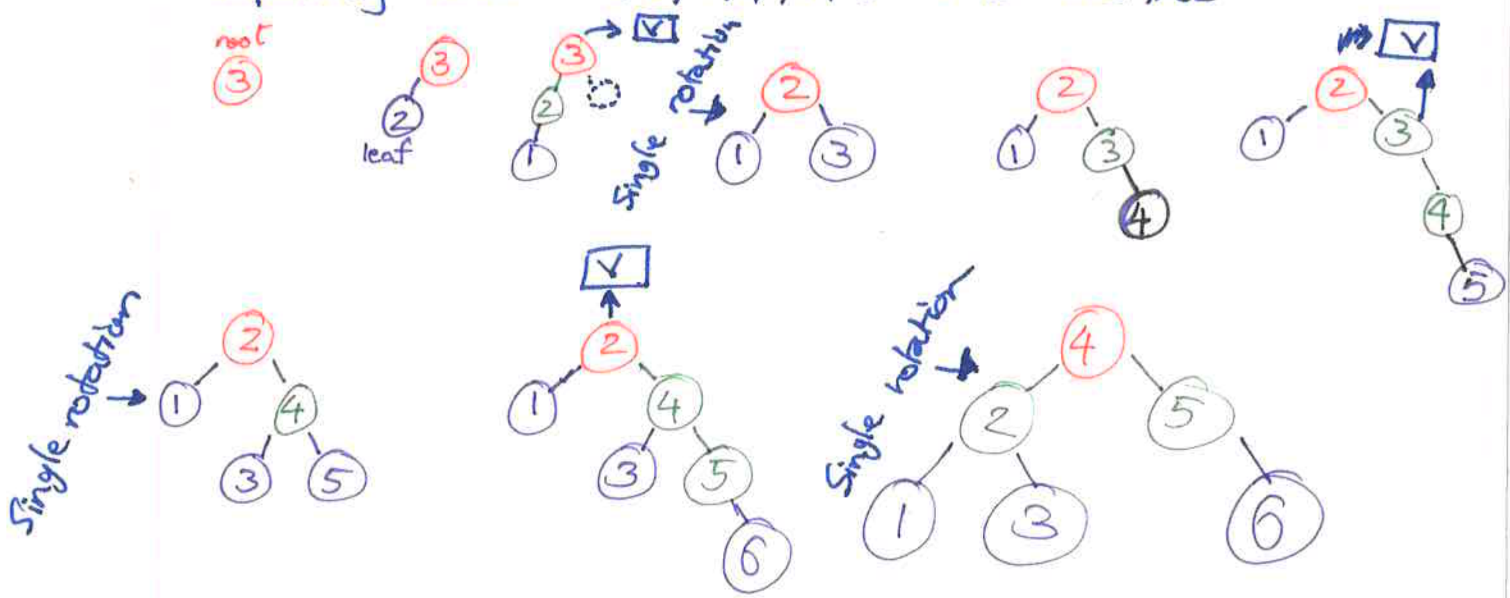
$\boxed{V}$: AVL violation at the ~~noted~~ node in question. i.e. $left(x) \neq right(x) \pm 1$ node

Sequentially insert 3, 2, 1, 4, 5, 6 to an AVL tree



End 25/June/2019 @ pg 4 Lecture 14 AVL

Jun 5th 2019

Single Right Rotation (SRR)

- → performed when A is unbalanced to left i.e. the left subtree of A is 2 higher than A's subtree
- → AND B is left heavy ; i.e. T1 is 1 higher than the right subtree of B T2

If both conditions are satisfied, SRR may be performed.

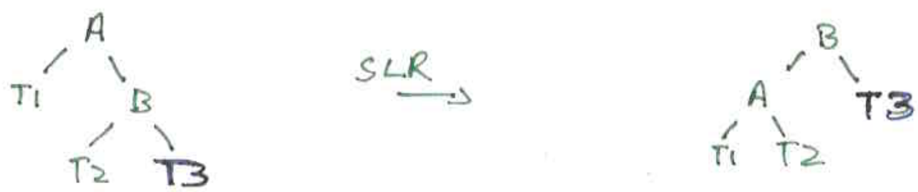
REVIEW o Cases during which the tree must be rebalanced

Let α denote the node that must be rebalanced

1) → Insertion into the left subtree of the left child of α.
2) → Insertion into the right subtree of the right child of α.
3) Insertion into the right subtree of the left child of α
4) Insertion into the left subtree of the right child of α.

addressed by 1 SRR 2 SLR

Single Left Rotation (SLR)

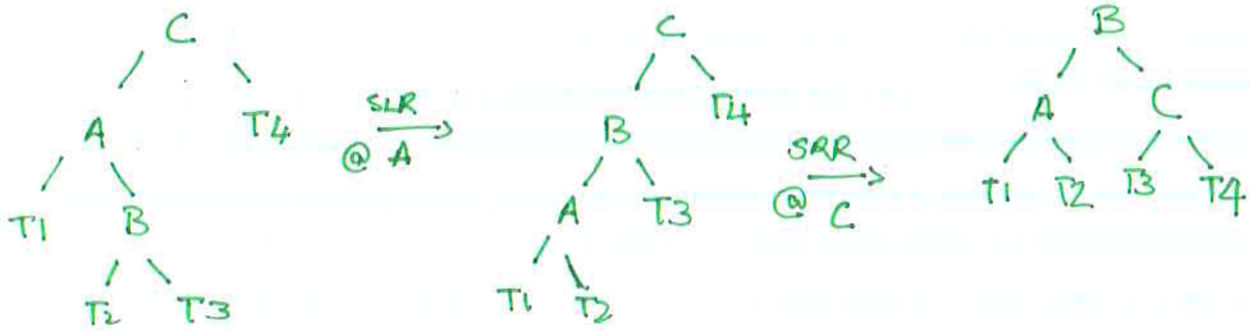


* → performed when A is unbalanced to the right

→ AND B is right-heavy i.e. T_3 is 1 higher than T2.

once these conditions are met, SLR may be performed

Double Left Rotation (DLR)



DLR = SLR + SRR (in that order) (different nodes)

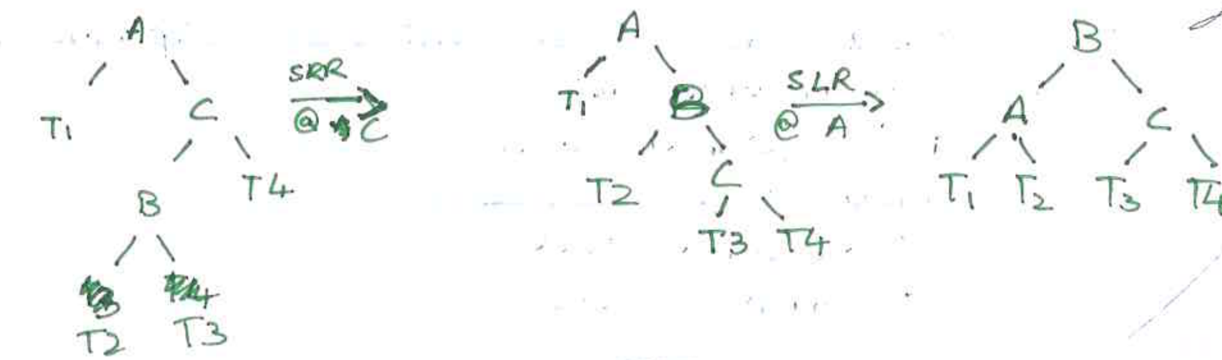
Performed when:

→ C is unbalanced to the left (left subtree is 2 more than the right subtree).

→ A is right heavy

→ Has one SLR node at A, following which it has one SRR node at C.

Double Right Rotation (DRR)



DRR = SRR + SLR (in that order) (different nodes)

Performed when:

-> A is unbalanced to the right (right subtree is 2 more than the left subtree)

-> C is ~~right~~ left heavy

-> Has one SRR at C, followed by an SLR at A.

%

B - trees

- The running time of ~~a~~ disk based algorithms can be measured in terms of:
  - CPU (computing time)
  - number of disk accesses
    - sequential reads
    - random reads

There is a need ~~of~~ for a search-tree structure that is secondary memory ~~enab~~ enabled

eg, to ensure we dont lose data in case of a power outage

eg. if the data structure becomes to memory intensive to viable use it without secondary storage.

Pointers in data structures are no longer addresses in main memory.

If x is a pointer to an object:

- if x is in main memory, then Key[x] refers to the object etc.
- In secondary memory:
  DiskRead(x) reads the object ~~fr~~ from disk into main memory (and DiskWrite(x) stores it back into the disk)

- The size of a B-tree's nodes is determined by the page size.

- A B-tree of height 2 can contain over 1 Billion keys.

- The heights of a binary-tree is usually a logarithm of base 2.
- But the heights of a B-tree is a logarithm of base (eg) 10000 etc.

i.e B-tree nodes have several orders of magnitude of children, even at height 2 (of the B-tree).

Definitions

- Node (x) has fields.
  - → $n[x]$: the number of keys of that node
  - → $key_1[x] < \cdots < key_{n}[x]$ $(n > 1)$
    i.e. the keys are arranged in ascending order.
  - → $leaf[x]$: returns true if the node is a leaf & false otherwise
  - → if x is an internal node then:
    $c_1[x], \ldots c_{n[x]+1}[x]$ are pointers to x's children with keys $\{1, 2 \ldots n[x]+1\}$
  - → leaf nodes have no children.

- Keys seperate the ranges of keys in the sub-trees.
  If k_i is an arbitrary key in the subtree $c_i[x]$, then
  $k_i <= key_i[x] \leq k_{i+1}$

- B-tree. Definitions (cont'd)

- Every tree has the same depth = the height of the tree = $\underline{\underline{h}}$

- In a B-tree of degree $\underline{\underline{t}}$:
  - → Every node other than the root must have at least $(t-1)$ keys.
  - → Every internal-node other than the root has at least "t" children.
  - → Every node may contain at most $2t-1$ keys.
    An internal node therefore has a maximum of "$2t$" children
  - → The root node has between 0 & $2t$ children
    (i.e. between 0 and $(2t-1)$ keys)

---

def: Height of a B-tree:

B-tree 'T' of height $\underline{h}$, containing $n \geq 1$ keys and a minimum degree $t \geq 2$, the following restriction on the height holds:

Imp

$$\boxed{h \leq \left[\log_t \left(\frac{n+1}{2}\right)\right]}$$

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

i.e $\boxed{n \geq 2t^h - 1}$

B-tree operations

- The following B-tree operations must be supported when implementing B-trees:
  - → Searching
  - → Creating an empty tree
  - → Insertion
  - → Deletion

- Searching:

```
BTreeSearch ( x , k ) :

    assign i = 1

    while (i <= n[x] && k > key_i[x]):
        assign i = i+1

    if (i <= n[x] && k = key_i[x]):
        return (x, i)

    if (leaf[x]):
        return NIL.
    else (DiskRead (c_i[x]):
        return BTreeSearch (c_i [x], k)
```

- Creating an Empty B-tree

→ simply "create a root" and write the newly created root to secondary-memory.

BTreeCreate (T):

assign x = AllocateNode(); // allocates disk space for node x (new node)

assign leaf[x] = ~~false~~ true. // set x's leaf property to true, since it is the first node (i.e it is a root & a leaf)

assign n[x] = 0; // The newly created x has no children ⇒ n[x] = 0 i.e the number keys @ x = 0 since no. of children @ x = 0

DiskWrite [x]

assign root[T] = x.

} // end fn.

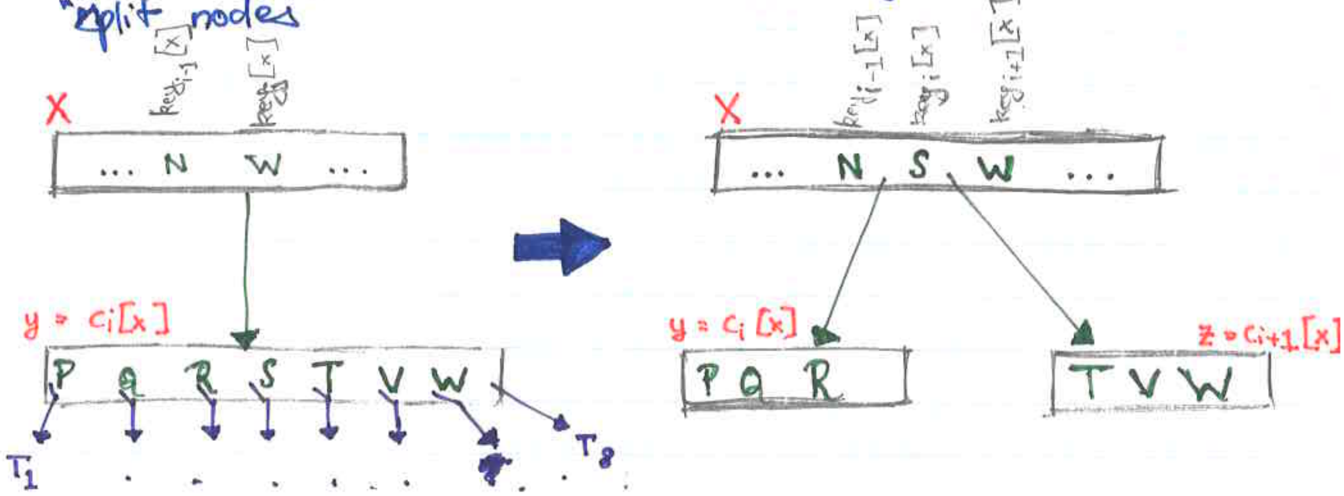
// the data here may now be written to disk.

// root[T] = x is done after DiskWrite(x) because "T" as an object is still in memory and is being manipulated. Each node in T has references to consecutive nodes and the "T" object is really just to use our data. We don't have to write "T" to disk

Splitting Node

→ Nodes fill up and reach their maximum capacity at $(2t - 1)$

→ Before we can insert another new key, we need to "split nodes"



from notes (Prof. LT) : (i) one key from "y" moves up to the parent node
(ii) the whole process results in two nodes each with $(t-1)$ keys (i.e approx $\frac{1}{2}$ of $(2t-1)$ keys from "y" prior to splitting the node)

Algorithm on next page

IMP

Splitting Nodes (algorithm)

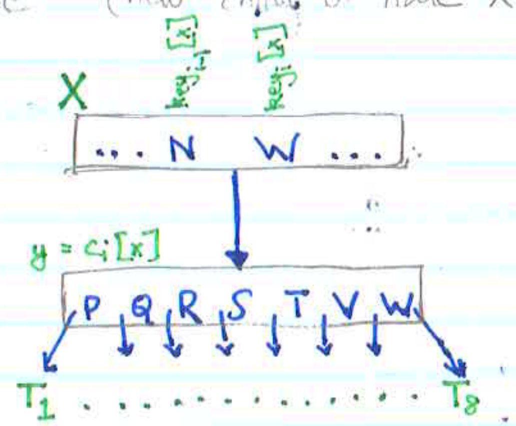
Some preliminary housekeeping before the algorithm:

→ x : parent node

→ y : child of x.
  ↳ node to be split

→ i : index in node x.

→ z : new node (new child of node x after "splitting node y")



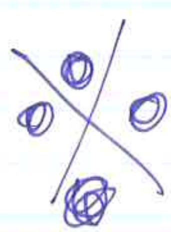
pseudo code:

```
BTreeSplitChild (x, i, y):
 · assign z = AllocateNode();

 · assign leaf[z] = leaf[y];

 · assign n[z] = t - 1;

 · for (j = 1; j <= t-1; j++){
     assign key_j[z] = key_{j+t}[z]
   }
```

con't'd
pseudocode for splitting a node

```
· if (! leaf[y]) {
      for (j=1; j<= t; j++) {
         assign c_j[z] = c_{j+t}[y]
      }
  }

· assign c_{i+1}[x] = z

[illegible]
· for (j = n[x]; j >= i; j--) {
      key_{j+1}[x] = key_j[x]
  }

* assign · key_i[x] = key_t[y]

* assign · n[x] = n[x] + 1

    · DiskWrite(y)
    · DiskWrite(z)
    · DiskWrite(x)

} // end function..
```

IMP



Running time of "Splitting node"

- Splitting a node is an operation that runs locally on the computer's memory and does not traverse the B-tree.
- $\Theta(t)$ [illegible] based on the specified degree: since two loops run t times.
- 3 input/output disk operations

XX ———— XX

(complex function bcoz B-tree) (66)

INSERTING KEYS TO A B-TREE

- Inserting a key to a B-tree is done recursively from the root all the way down recursively to the leaf level.

- Before descending to a lower level in a B-tree, we must make sure that the node has $< (2t-1)$ keys.

  → Also recognise that if a node is full (edge/special case) then we split the node ~~before inserting~~ (in this case the root node of B-Tree T) ~~using~~ as described below in the pseudocode. and set T's new root as the ~~the~~ newly created ~~root node~~ "overflow node" s.

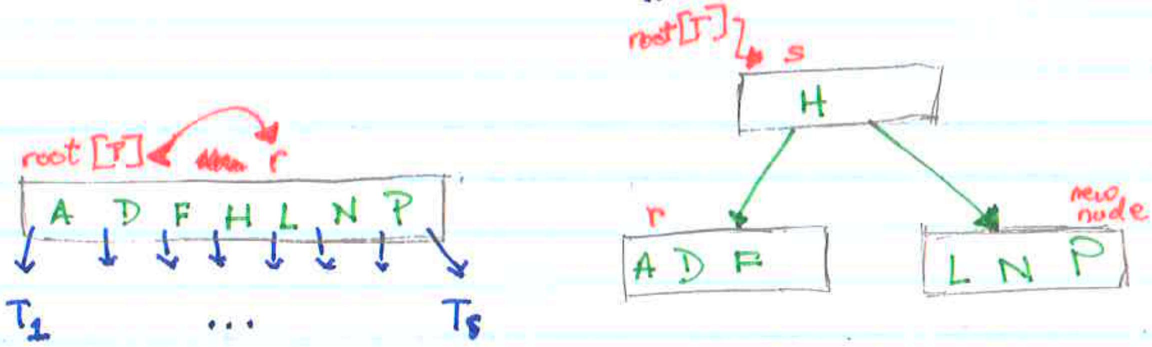
Pseudo code:

```
BTreeInsert (T, k) {
  · assign r = root[T];

  · if (n[r] == (2t-1)) {
      assign s = AllocateNode()
      assign root[T] = s
      assign leaf[s] = False
      assign n[s] = 0
      assign C1[s] = r

      BTreeSplitChild (s, 1, r)
      BTreeInsertNonFull (s, k)

  else : BTreeInsertNonFull (r, k)
} //end function.
```

// BTreeInsertNonFull attempts insert a key k into a node (x) under the assumption that x is not full (ie. $n[x] < (2t-1)$)

// BTreeInsert and the recursion in BTreeInsertNonFull guarantees that the check is done accurately and the assumption above holds ~~for~~ true j

- **Splitting the B-tree's root**



→ To split the, we need to create new nodes

→ The B-tree grows at the top instead of the bottom when splitting the root

pseudo code refferred to in BTreeInsert()

- BTreeInsertNonFull (x, K):

```
· assign i = n[x]

· if (leaf[x]) {
    while ((i >= 1) && (k < key_i[x])) {
        key_{i+1}[x] = key_i[x]
        i = i-1
    }
    key_{i+1}[x] = k
    n[x] = n[x] + 1
    DiskWrite(x)

· else ...
```

leaf insertion

(cont'd from prev. page) else {

```
while ((i >= 1) && (k < key_i[x])) {
    i = i - 1
}

i = i + 1
DiskRead ( c_i[x] )

if ( n [c_i[x]] == (2t - 1)) {
    BTreeSplitChild (x, i, c_i[x])
    if (k > key_i[x]) {
        i = i + 1
    }
}

BTreeInsertNonFull (c_i[x], k)
} //end else
} // end function
```

internal node. code traverses the B-tree

Insertion running time

→ Disk I/O: $O(h)$, since disk access in the code are $O(1)$ and are performed in recursive calls of BTreeInsertNonFull()

→ CPU: $O(t * h) = O(t * \log_t n)$

→ At any given moment, there are $O(1)$ number of disk pages in main memory end July 5th ↑↑

Lecture 16
B-trees part A (prof LT's notes)

↳ good work today.
was super productive. Try & do some more.