NYU, Tandon School of Engineering

November 20, 2024

CS-GY 6083

Principles of Database Systems Section A, Fall 2024

Homework #5

Submitted by:

Khwaab Thareja

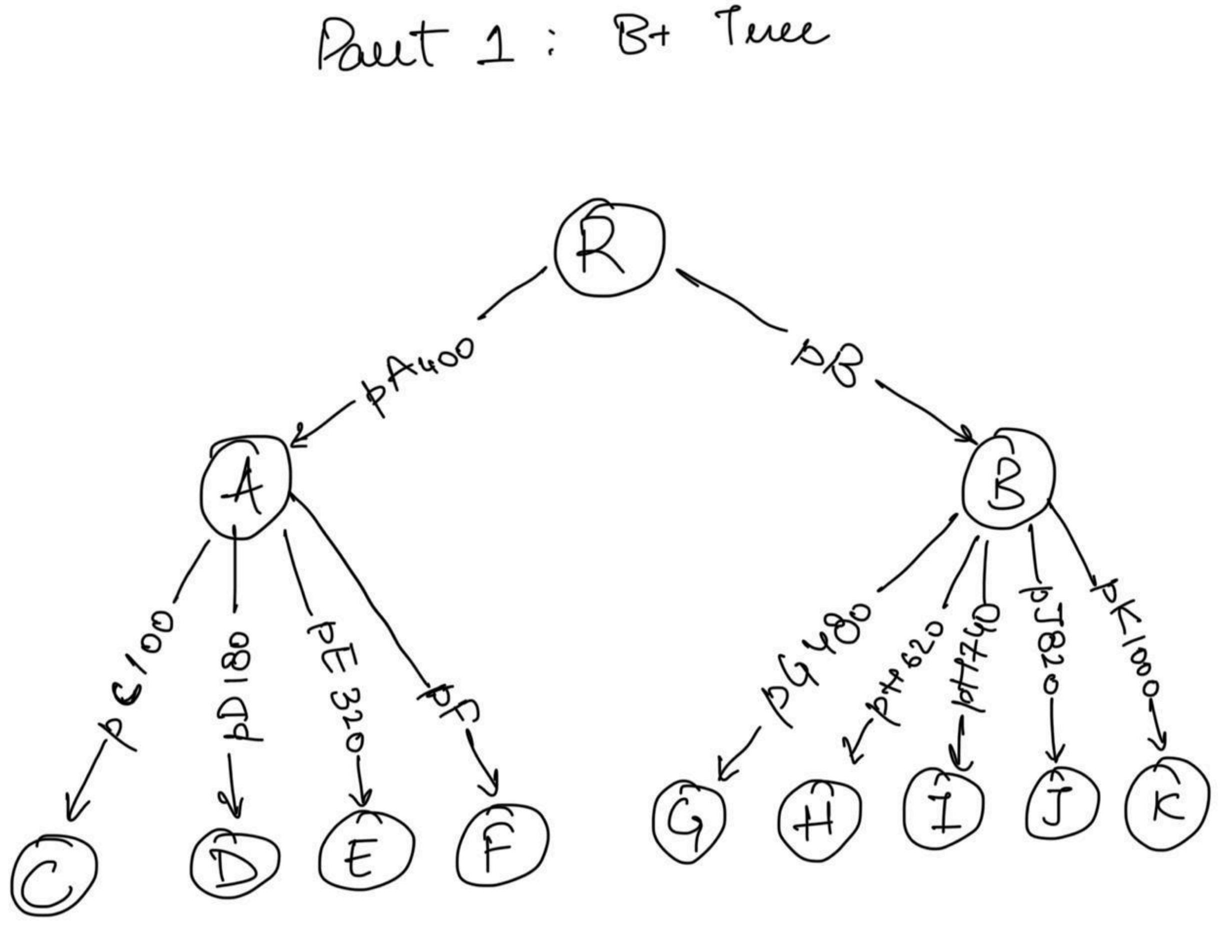
N15911999

Kt3180

Guided By: Prof Phyllis Frankl

Answer 1

We will start by visualising the B+ Tree which will make it easier to understand the flow to find the key.



Answer 1: Part 1

a. Find the record with key value 680

We would first visit the node R which has a key value of 400. We further compare 680 with 400 and as 680>400, we proceed to pointer B

Now we compare 680 with key values present in B such as 480, 620 and 740 and find that 620<680<740 , which belongs to pointer I. therefore we follow pointer I and find 680

1. **Checking which all nodes are visited:**
    - **R**: Compare 680 with 400 → Move to child **B**.
    - **B**: Compare 680 with 480, 620, 740 → Move to child **I**.
    - **I**: Compare 680 with 620, 640, 660 → Find 680.
2. **Comparison for key values:**
    - At **R**: Compare 680 > 400.
    - At **B**: Compare 680 > 480, 680 > 620, 680 < 740.
    - At **I**: Compare 680 > 620, 680 > 640, 680 > 660, and find 680.
3. **Data fetched:** Record with key value 680.

Now we use this pointer before 680 and fetch the record with key 680 from the data file.

b. Find the record with key value 620

We would first visit the node R which has a key value of 400. We further compare 620 with 400 and as 620>400, we proceed to pointer B

Now we compare 620 with key values present in B such as 480, 620 and 740 and find that 620=620<740 , which belongs to pointer I. therefore we follow pointer I and find 620

1. **Checking which all nodes are visited:**
    - **R**: Compare 620 with 400 → Move to child **B**.
    - **B**: Compare 620 with 480, 620 → Move to child **I**.
    - **I**: Compare 620 → Find 620.
2. **Comparison for key values:**
    - At **R**: Compare 620 > 400.
    - At **B**: Compare 620 > 480, 620 = 620.
    - At **I**: Find 620 immediately.

3. **Data fetched:** Record with key value 620.

Now we use this pointer before 620 and fetch the record with key 680 from the data file.

c. Find all records with key values between 685 and 825 (inclusive)

We would first visit the node R which has a key value of 400. We further compare the lower value 685 with 400 and as 685>400, we proceed to pointer B

Now we compare the lower value 685 and 825 with key values present in B such as 480, 620 and 740 and find that 620<685<740 and 820<825<880 , so we have values belonging to pointer I to K. therefore we follow pointer I and find values till we find the last value 820 in pointer K.

1. **Checking which all nodes are visited:**
   - **R**: Compare 685 with 400 → Move to child **B**.
   - **B**: Compare 685 with 480, 620, 740 → Visit children **I**, **J**, **K**.
2. **Comparison for key values:**
   - At **R**: Compare 685 > 400.
   - At **B**: Compare 685 > 480, 685 > 620, 685 < 740 → Move to **I**, **J**, and **K**.
   - At **I**: Collect keys 685–720.
   - At **J**: Collect keys 740–800.
   - At **K**: Collect key 820.
3. **Data fetched:**
   - Records with key values: 700, 720 (from **I**), 740, 760, 780, 800 (from **J**), and 820 (from **K**).

So Comparisons are made with 700 as the first value in the range and 840 as the last value. Key values are directly mapped to physical order in data file so no additional lookup required.

d:

We would first visit the node R which has a key value of 400. We further compare the lower value 685 with 400 and as 688>400, we proceed to pointer B

Now we compare the lower value 685 and 825 with key values present in B such as 480, 620 and 740 and find that 620<680<740 and 820=820<880 , so we have values belonging to pointer I to K. therefore we follow pointer I and find values till we find the last value 820 in pointer K.

The data file is not organized in the same order as the index. For each key found in the index, a separate lookup in the data file is required.

Steps:

1. **Traversal to locate relevant keys (680–820):**
   - **R**: Compare 680 with 400 → Move to child **B**.
   - **B**: Compare 680 with 480, 620, 740 → Visit children **I**, **J**, **K**.
   - **I**: Keys [680, 700, 720] fall in range.
   - **J**: Keys [740, 760, 780, 800] fall in range.
   - **K**: Key [820] falls in range.
2. **Checking which all nodes are visited:**
   - Root (**R**) → Internal node (**B**) → Leaf nodes **I**, **J**, **K**.
3. **Comparison for key values:**
   - At **R**: Compare 680 > 400.
   - At **B**: Compare 680 > 480, 680 > 620, 680 < 740.
   - At **I**, **J**, **K**: Collect keys within range [680–820].
4. **Data fetched:**
   - For each key in the range [680, 700, 720, 740, 760, 780, 800, 820], perform **individual lookups** in the data file, as the data is not stored sequentially.

In this case, the index records points to a bucket that point to actual records

Answer 1: Part 2

a. Insert 795

We would first visit the node R which has a key value of 400. We further compare 795 with 400 and as 795>400, we proceed to pointer B

Now we compare 795 with key values present in B such as 620, 740 and 820 and find that 740<795<820 , which belongs to pointer J. Therefore we proceed to insert 795 to node J.

1. **Locate Leaf Node:**
    - Traverse to B → Compare 795 with 480, 620, 740, 820 → Move to J.
    - Keys in J: [740, 760, 780, 800].
2. **Insert 795 into Leaf Node J:**
    - Keys in J after insertion: [740, 760, 780, 795, 800].
    - No overflow occurs (leaf nodes can hold up to 7 keys).
3. **Changes:**
    - Updated keys in J: [740, 760, 780, 795, 800].

Since there is enough space , we can directly add 795 to the leaf node and no splitting is required.

Hence after insertion we have J: [740, 760, 780, 795, 800]

b. Insert 192

We would first visit the node R which has a key value of 400. We further compare 192 with 400 and as 192<400, we proceed to pointer A

Now we compare 192 with key values present in A such as 100,180 AND 320 and find that 180<192<320 , which belongs to pointer E. Therefore we proceed to insert 192 to node E.

1. **Locate Leaf Node**:
    - Traverse to **A** → Compare 192 with 100, 180, 320 → Move to **E**.
    - Keys in **E**: [180, 200, 220, 240, 260, 280, 300].
2. **Insert 192 into Leaf Node E**:
    - Keys in **E** after insertion: [180, 192, 200, 220, 240, 260, 280, 300].
    - Overflow occurs (leaf nodes can hold only 7 keys).
3. **Split Leaf Node E**:
    - **E**: [180, 192, 200, 220].
    - New node **E'**: [240, 260, 280, 300].

    - Update parent (**A**) to include the separator key (240).
4. **Changes**:
    - **E**: [180, 192, 200, 220].
    - **E'**: [240, 260, 280, 300].
    - **A**: [100, 180, 240, 320].

Since a leaf node can only hold 7 Keys, and Node E already had 7 keys before insertion of 192, hence splitting is required, hence a new internal node **E'** is added. Also, A is updated with a separator key.

c. Insert 593

We would first visit the node R which has a key value of 400. We further compare 593 with 400 and as 593>400, we proceed to pointer B

Now we compare 593 with key values present in B such as 480 AND 640 and find that 480<593<640 , which belongs to pointer H. Therefore we proceed to insert 593 to node H.

1. **Locate Leaf Node**:
    - Traverse to **B** → Compare 593 with 480, 620 → Move to **H**.
    - Keys in **H**: [480, 500, 520, 540, 560, 580, 600].
2. **Insert 593 into Leaf Node H**:
    - Keys in **H** after insertion: [480, 500, 520, 540, 560, 580, 593, 600].
    - Overflow occurs (leaf nodes can hold only 7 keys).
3. **Split Leaf Node H**:
    - **H**: [480, 500, 520, 540].
    - New node **H'**: [560, 580, 593, 600].
    - Update parent (**B**) to include the separator key (560).
4. **Changes**:
    - **H**: [480, 500, 520, 540].
    - **H'**: [560, 580, 593, 600].
    - **B**: [480, 560, 620, 740].

Since a leaf node can only hold 7 Keys, and Node H already had 7 keys before insertion of 593, hence splitting is required, hence a new internal node **H'** is added. Also, B is updated with a separator key.

d. Delete 200

We would first visit the node R which has a key value of 400. We further compare 200 with 400 and as 200<400, we proceed to pointer A

Now we compare 200 with key values present in A such as 100,180 AND 320 and find that 180<200<320 , which belongs to pointer E. Therefore we proceed to delete 200 from node E.

1. **Locate Leaf Node:**
    - Traverse to A → Compare 200 with 100, 180, 320 → Move to E.
    - Keys in E: [180, 200, 220, 240, 260, 280, 300].
2. **Remove 200 from Leaf Node E:**
    - Keys in E after deletion: [180, 220, 240, 260, 280, 300].
    - No underflow occurs (leaf nodes must have at least 4 keys).
3. **Changes:**
    - Updated keys in E: [180, 220, 240, 260, 280, 300].

Since leaf node has greater than 4 keys after deletion, hence no underflow occurs.

Hence, No splits, merges, or redistribution needed since the node remains within valid limits.

e. Delete 340

We would first visit the node R which has a key value of 400. We further compare 340 with 400 and as 340<400, we proceed to pointer A

Now we compare 340 with key values present in A such as 320 and find that 320<340 , which belongs to pointer f. Therefore we proceed to delete 340 from node F.

Locate Leaf Node:

- Traverse to A → Compare 340 with 100, 180, 320 → Move to F.
- Keys in F: [320, 340, 360, 380].

Remove 340 from Leaf Node F:

- Keys in F after deletion: [320, 360, 380].
- Underflow occurs (less than 4 keys).
- Merge F with sibling E .

After Merge:

- Merged Node: [180, 200, 220, 240, 260, 280, 300, 320, 360, 380].
- This exceeds the maximum of 7 keys, causing an overflow.
- Split into two balanced nodes:
  - **New Node E**: [180, 200, 220, 240, 260].
  - **New Node F**: [280, 300, 320, 360, 380].
- Update parent **A** with the new separator key (280).

Parent Node Update:

- **A**: [100, 180, 280, 320].

As we delete 340 from node F, it is left with 3 keys which initiated an underflow condition and merges with E. Now node E is left with 10 Keys which initiates the overflow condition and thus splitting is required which splits node E and F with 5 keys each and updates A with a new separator.

f. Delete 40

We would first visit the node R which has a key value of 400. We further compare 40 with 400 and as 40<400, we proceed to pointer A

Now we compare 40 with key values present in A such as 100 and find that 40<100 , which belongs to pointer C. Therefore we proceed to delete 340 from node F.

1. **Locate Leaf Node:**
   - Traverse to A → Compare 40 with 100 → Move to C.
   - Keys in C: [20, 40, 60, 80].
2. **Remove 40 from Leaf Node C:**
   - Keys in C after deletion: [20, 60, 80].
   - Underflow occurs (leaf nodes must have at least 4 keys).
3. **Merge C and D:**
   - Merge C and D to form a new leaf node:
     - Merged Node: [20, 60, 80, 100, 120, 140, 160].
   - Remove C and the separator key 100 from parent node A.
   - Keys in A after update: [180, 320].
   - A now has only 2 children, causing an underflow.
4. **Handle Underflow in A:**
   - Redistribute with sibling B:
     - Move 480 from B to A as a separator key.
     - Move H (the pointer to the first child of B) from B to A.
5. **After Redistribution:**
   - A: [180, 320, 400,480] 5 children: [C+D (merged), E, F, G,H]).

- B: [740, 820…] (6 children: [I, J, K, L, M, N]).
- Root R : [620].

After deleting 40 from Node C, It went in underflow state due to which C and D had to be merged due to which A went into Underflow condition and had to be merged with B. However, Together A-B merged together went into overflow condition and had to be split.

Answer 2. Part 1

A. Determine the Largest Value of n.

Given:

- Block size: 4 KB=4096 bytes4
- Each node stores n−1 and n pointers.
- Size of a key (sID): 8 bytes8
- Size of a pointer (RID): 8 bytes

Node size:

Node size=(n−1) · key size+n · pointer size

Node size=(n−1) · 8+n · 8=8n−8+8n=16n−8

Fit into block:

16n−8≤4096

16n≤4104

n≤410416 → 256.5

The largest value of n is:

n=256

Thus:

- **Each node can store 255 keys and 256 pointers.**

Part B:

Let us Assume:

1. Total records in **Takes** = 500,000,000
2. Each key is associated with one pointer in the leaf nodes.
3. x, the number of children per node, is close to n/2 to n. Choose x=200 for estimation.

Each leaf node stores n−1=255keys.

Leaf nodes=⌈total keys / per leaf node⌉

Leaf nodes=⌈500,000,000/255⌉=1,960,785

Each internal node has x=200 children.

- At each level:

Nodes at level=⌈nodes at previous leve / x ⌉

1. **Leaf level:** 1,960,785
2. **Next level:** ⌈1,960,785 / 200⌉=9,804
3. **Next level:** ⌈9,804 / 200⌉=50
4. **Root level:** ⌈50 / 200⌉=1.

Therefore,

Height=4 (leaf level + 3 internal levels)

Part C

- Lets take H=4, so searching the B+ Trees Require 4 block transfers and 4 seek for traversal.
- Lets assume there are no buckets
- The student has to 10 records which fits in 1 block or leaf node.
- In non clustering, the 10 records might be present in 10 different blocks, one for each record.
- So 10 additional fetch blocks and seeks required.
- Total block transfers = 4+10 = 14 ; Total Seek = 10+ 4 = 14

Part D

A. Since its clustered, the data must be physically ordered by sID,

   Leaf nodes must point to **sequential blocks** in the Takes file, with all records for the same sID stored contiguously.

B. No, the **height of the tree** would remain the same because the B+ tree structure depends on the total number of records and nnn, not on clustering.

C. For a clustering index, after locating the first record, subsequent records are fetched sequentially from the disk.

   Searching the tree will be same as for the non-clustering index: 4 block transfers + 4 seeks4

   Finding records wil have All 10 records are stored contiguously, so they can be fetched with:

   i. 1 block transfer.
   ii. 1 seek.

   Therefore, Total cost with clustering index:

   - Total block transfers=4+1=5
   - Total seeks=4+1=5