CS554, Homework 4

- **Question 1 (20 pts)**
  - **Exercise 14.1.1 (a)**

    - Suppose blocks hold either three records, or ten key-pointer pairs.

      As a function of **n** (= the number of records), how many blocks do we need to store a data file using a **dense index**

    Answer:

  - **Exercise 14.1.1 (b):**

    - Suppose blocks hold either three records, or ten key-pointer pairs.

      As a function of **n** (= the number of records), how many blocks do we need to store a data file using a **sparce index**

    Answer:

- **Question 2 (15 pts - each subquestion 5 points)**

The nodes B-tree in following questions have a maximum of **3 keys and 4 pointers**.

1. Show the pointers followed by the operation to lookup the record with key 41: (don't forget to include the **data record pointers** -- i.e., the bottom arrows)

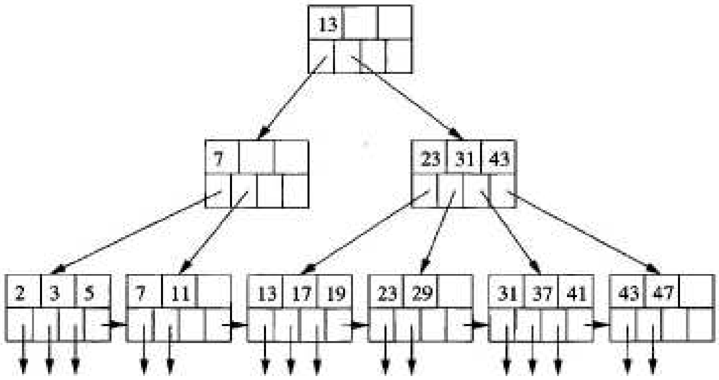


Figure 14.13: A B-tree

2. Show the pointers followed by the operation to lookup all records in the range 20 to 30: (don't forget to include the **data record pointers** -- i.e., the bottom arrows)

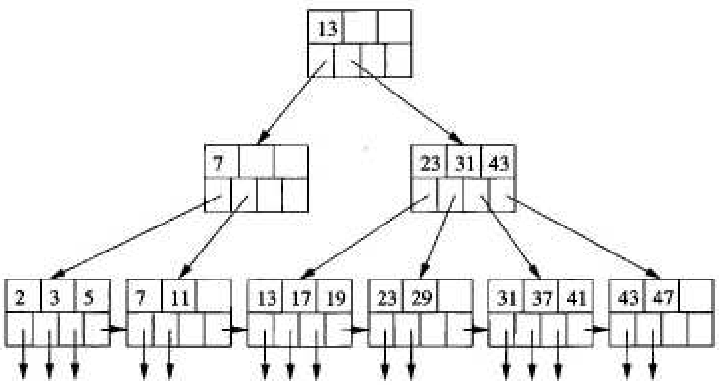
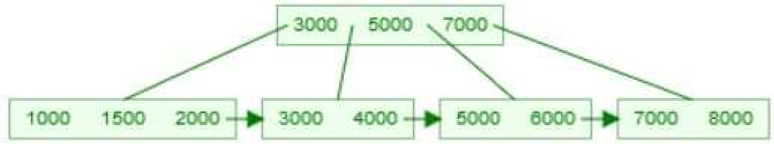


Figure 14.13: A B-tree

3. Starting with the following B-tree:



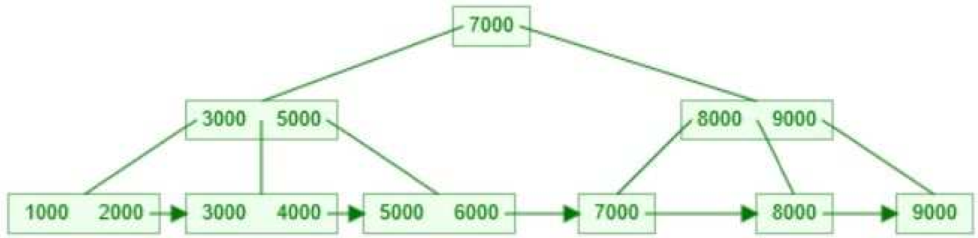
Show the B-tree after we insert a record with key **2500**:

- **Question 3 (15 pts - each subquestion 5 points)**

  The nodes B-tree in following questions have a maximum of **2 keys and 3 pointers**.

---

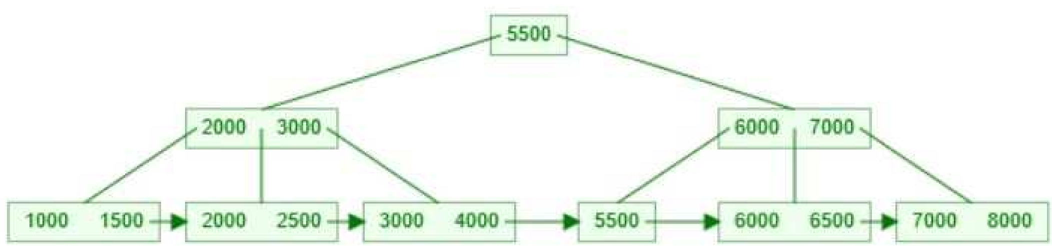
1. Starting with the following B-tree:



Show the B-tree after we delete the record with key **7000**

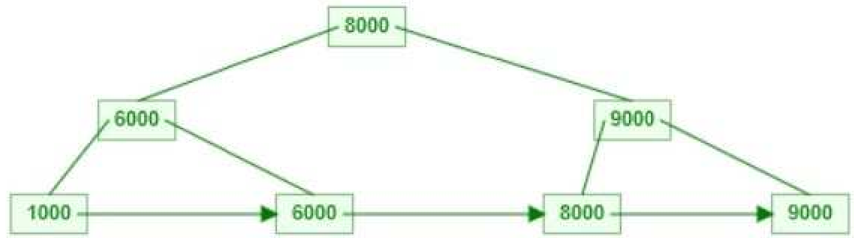
---

2. Starting with the following B-tree:



Show the B-tree after we delete the record with key **5500**

3. Starting with the following B-tree:



Show the B-tree after we delete the record with key **6000**

- **Question 4 (20 pts)**
  - **Consider** the **delete algorithm** for an ***internal* node** of the **B⁺-tree**:

```
/* ========================================================
   Delete( x, rx ) from a node N in B+-tree
   ======================================================== */
Delete( x, rx, N )
{
   Delete x, rx from node N;

   /* ====================================
      Check for underflow condition...
      ==================================== */
   if ( N has ≥ ⌊(n+1)/2⌋ pointers /* At least half full*/ )
   {
      return;    // Done
   }
   else
   {
      /* ---------------------------------------------------------
         N underflowed: fix the size of N with transfer or merge
         --------------------------------------------------------- */

      /* ========================================================
         Always try transfer first !!!
         (Merge is only possible if 2 nodes are half filled)
         ======================================================== */
      if ( leftSibling(N) has ≥ ⌊(n+1)/2⌋ + 1 pointers )
      {
         1. transfer last key from leftSibling(N) through parent into N
            as the first key;

         2. transfer right subtree link into N as the first link
      }
      else if ( rightSibling(N) has ≥ ⌊(n+1)/2⌋ + 1 pointers )
      {
         1. transfer first key from rightSibling(N) through parent into N
            as the last key;

         2. transfer left subtree link into N as the last link
      }

      /* ==========================================================
         Here:     can't solve underflow with a transfer

         Because: BOTH sibling nodes have minimum # keys/pointers
                  (= half filled !!!)

         Solution: merge the 2 half filled nodes into 1 node
         ========================================================== */
      else if ( leftSibling(N) exists )
      {
        /* ===============================================
           merge N with left sibling node
           =============================================== */
         1. Merge (1) leftSibling(N) + (2) key in parent node + (3) N
            into the leftSibling(N) node;

         2. Delete ( transfered key, right subtree ptr, parent(N) );   // Recurse !!
      }
      else // Node N must have a right sibling node !!!
      {
        /* ===============================================
           merge N with right sibling node
           =============================================== */
         1. Merge (1) N + (2) key in parent node + (3) rightSibling(N)
            into the node N;

         2. Delete ( transfered key, right subtree ptr, parent(N) ); // Recurse !!
      }
}
```

Currently, the **delete algorithm** for **internal node** does ***not*** handle a **deletion** in the ***root* node**

Question:

- **Add code** into the **above algorithm** that will **include support** for **deletion** in the **root node** of a **B^+-tree**

  Specify:

  1. **Where** the **code** will bbe **added** (**i.e.:** give the **position** in the **program** listed in the **previous figure**) (5 pts)

     ---

  2. The **code (= statements)** that you will need **add** to **handle** deletion in the **root node**: (15 pts)

- **Question 5 (30 pts)**
  - **Exercise 14.5.5:**
    - Suppose we store a relation R (x,y) in a grid file.
    - Both attributes have a range of values from 0 to 1000.
    - The partitions of this grid file happen to be uniformly spaced:
      - for x there are partitions every 20 units, at 20, 40, 60, and so on,
      - for y the partitions are every 50 units, at 50, 100, 150, and so on.

Questions:

- How many buckets do we have to examine to answer the range query

```
SELECT * FROM R
WHERE 310 < x AND x < 400
  AND 520 < y AND y < 730;
```

Answer:

- We wish to perform a **nearest-neighbor** query for the point **(110,205)** (i.e., find the **closest element** to the point (110,205))

  We begin by searching the bucket with lower-left corner at (100,200) and upper-right corner at (120,250).

  We indicate this bucket as:

```
[100-120][200-250]
```

  We find that the closest point in this bucket is **(115,220)**.

  What other buckets must be searched to verify that this point is the closest?

  Answer: