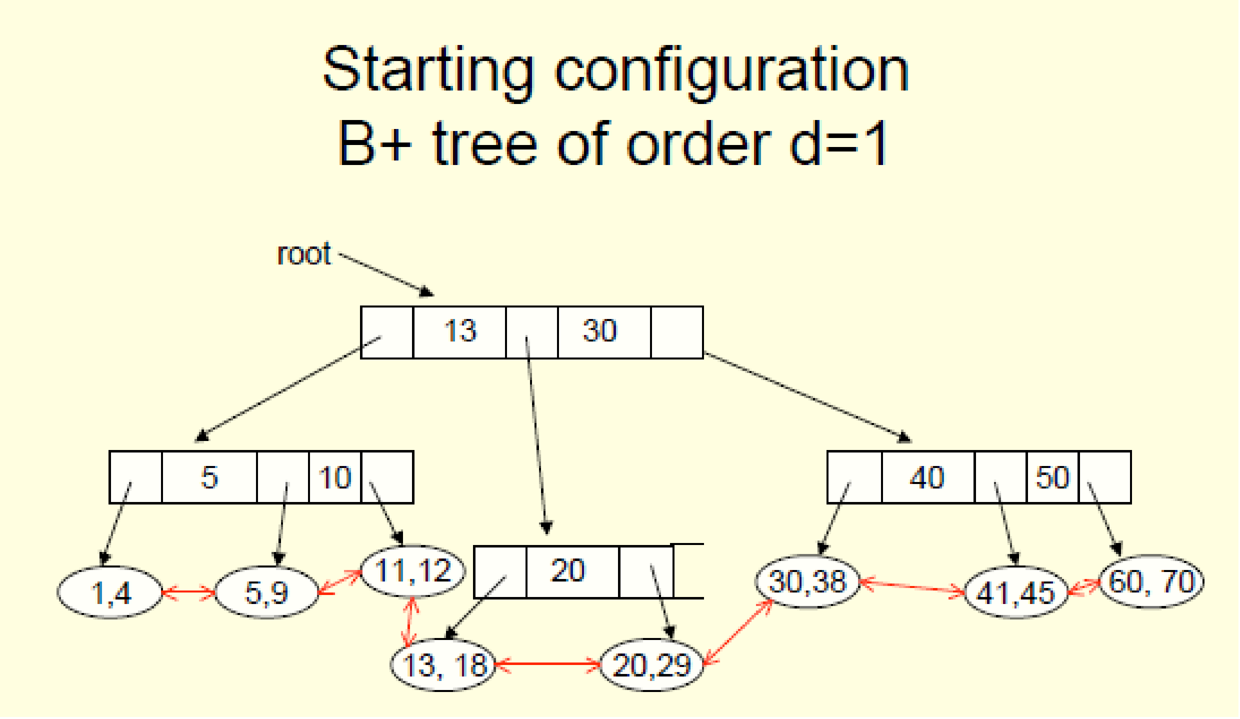
Testing Mechanism for B+ Trees

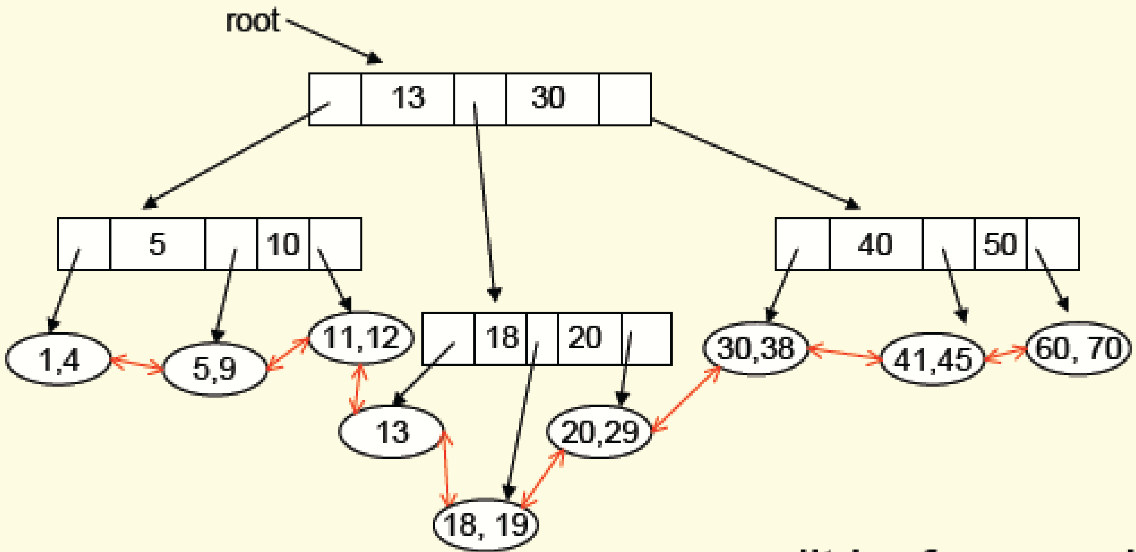
- 以下幾個練習題，希望能夠幫助大家想想怎麼去測試你們所建構的 **B+ Trees**。
- 它們的資料結構跟你們的 **B+ Trees** 或許並不完全相同。
- 練習題純粹做為參考，並沒有任何實作的強迫性。

Exercise 1:

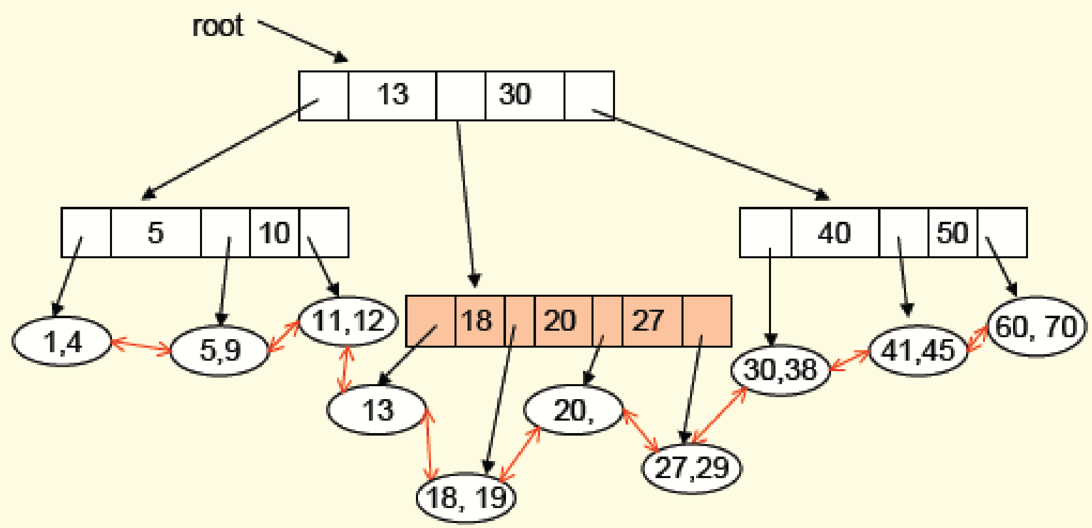


Testing B+ Trees (2)

Insert 19:
split leaf; expand parent with key 18

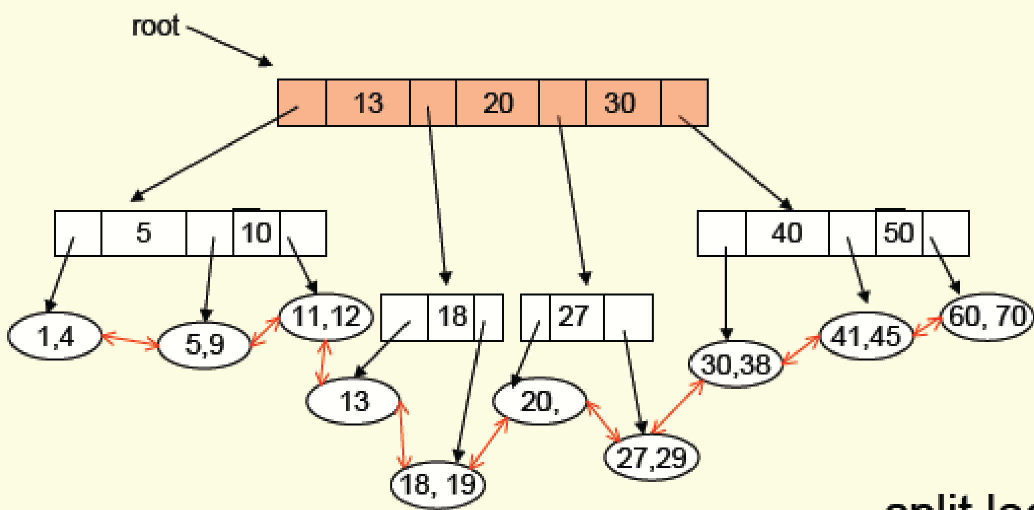


Insert 27
split leaf; expand parent with key 27 => too full

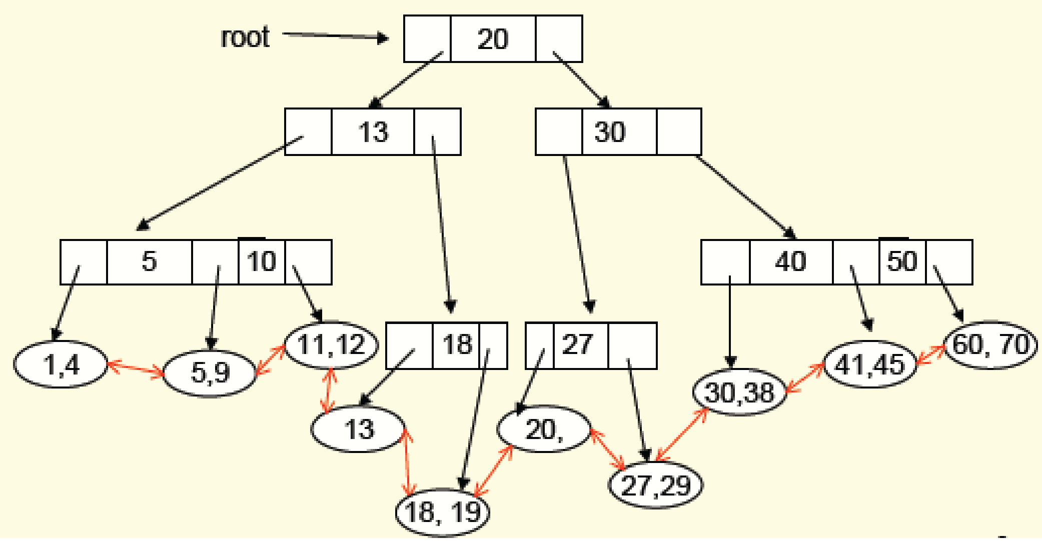


Testing B+ Trees (3)

Insert 27
split leaf; split parent;
expand grandparent with key 20 => too full

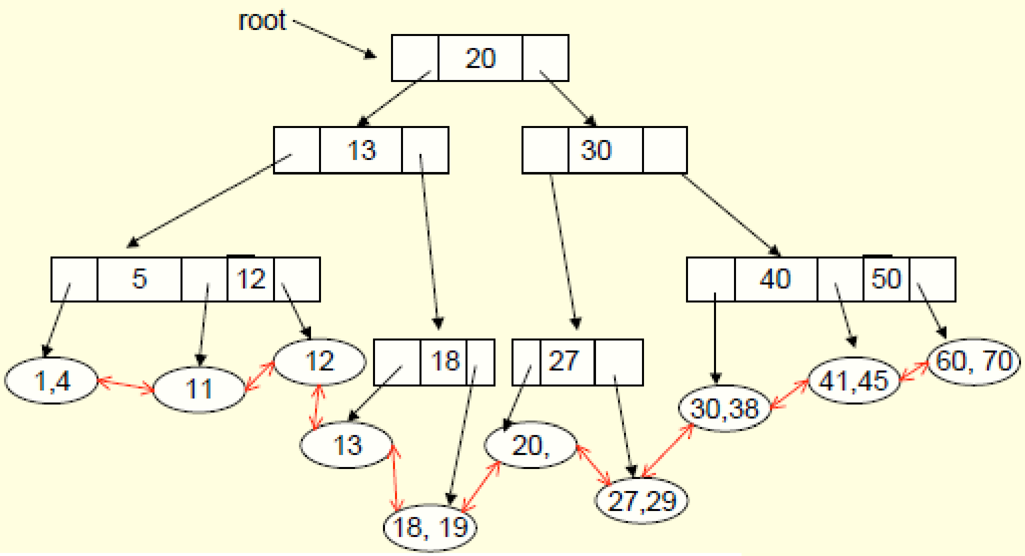


Insert 27
split leaf; split parent; split grandparent
new root with key 20

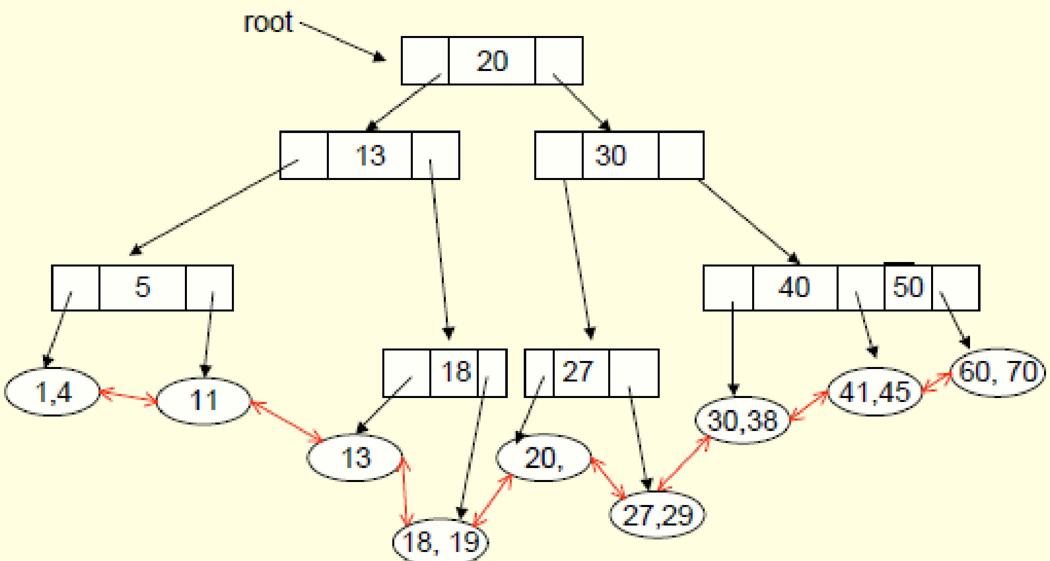


Testing B+ Trees (4)

Delete 5, then 9
redistribute from right sibling



Delete 12
merge leaves, delete key from parent

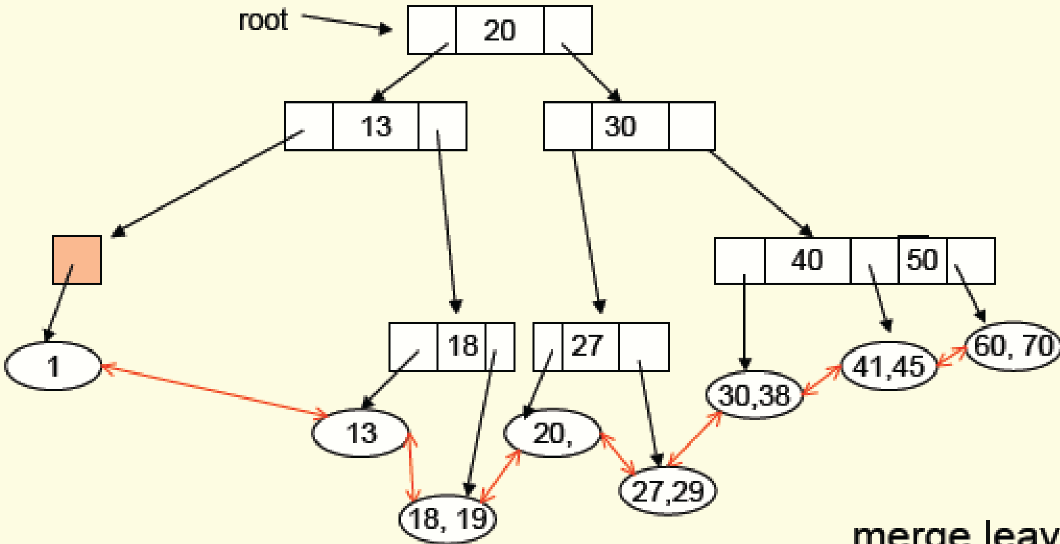


Testing B+ Trees (5)

Delete 4, then 11

merge leaves, delete key from parent

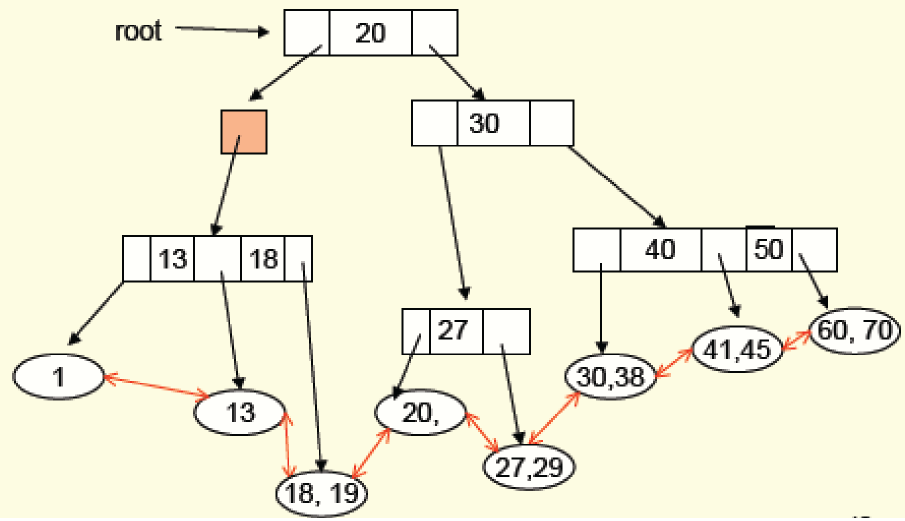
=>parent not full enough



Delete 4, then 11

merge leaves, merge parent, bringing down key 13

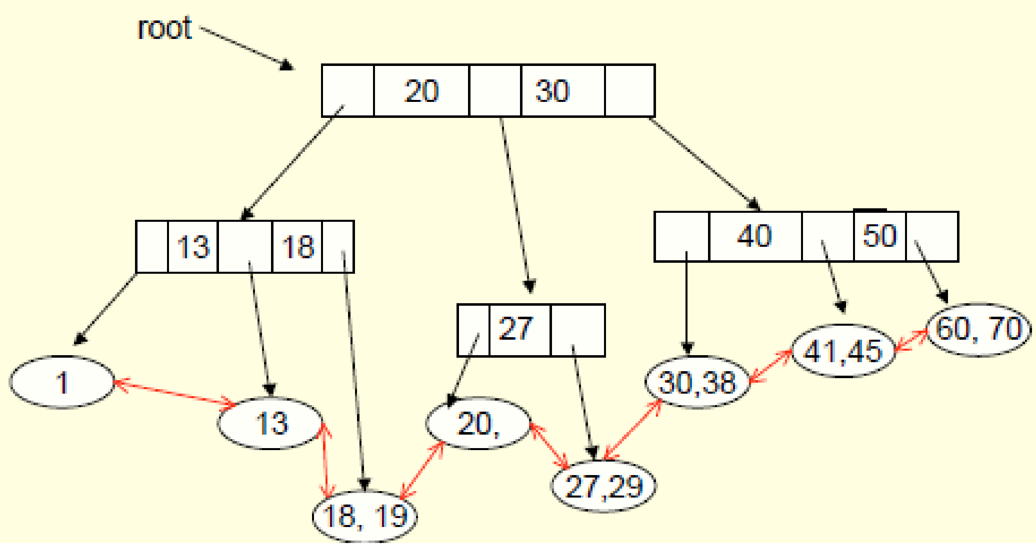
=>grandparent not full enough



Testing B+ Trees (6)

Delete 4, then 11

merge leaves; merge parent, bringing down key 13
merge grandparent, bring down key 20,
remove root



Testing B+ Trees (7)

Exercise 2:

- Assume this problem deals with the creation and use of a B+ Tree of order d=1. The leaves hold a maximum of 2 data entries.

Step 1. Bulk load a B+ Tree with data entries with keys values 2, 4, 6, 8, 10, 12, 14, 16 so that each leaf is full (or half-full). Show the state of the tree.

Step 2. Insert a data entry with key value 5 in your B+ Tree of Step 1. Show the state of the tree after the execution of the insert algorithm.

Step 3. Delete the data entry with key value 12 from your B+ Tree of Step 1. Show the final tree.

Testing B+ Trees (8)

Exercise 3:

- Initialize a B+ tree of order 2. Make your program insert the following integers, in this order:

  2, 16, 7, 20, 29, 33, 39, 38, 3, 5, 14, 19, 22, 24, 27, 34

- Insert 23 into the B+ Tree. Show the state of the tree.
- Delete 29, 33, and 34 from the B+ Tree. Show the final tree.

Testing B+ Trees (9)

- 以下定義適用於 **Exercise 4** 至 **Exercise 6**。

- number of search keys = x -> number of pointers $p = x+1$
- number of pointers p -> search keys $d = p–1$
- Each node that is not a root or a leaf has between $p/2$ and p children, where p is fixed for a particular tree.
- A leaf node has between $(p-1)/2$ and $p-1$ values (search keys)
- Size of one node = $2x + 1 = x + p$

Testing B+ Trees (10)

Exercise 4:

- Consider a relational table R consisting of 12 tuples with primary key values 40, 20, 10, 30, 15, 55, 50, 21, 3, 54, 2, 8. Perform the following operations:

- Index it with a B+ tree with $x = 3$ (number of search keys) on the primary key attribute. Insert the tuples in the above-specified order. How does the resulting tree look like?
- Delete values 7, 21, 50 from the index.

Testing B+ Trees (11)

Exercise 5:

- Consider a relational table R consisting of the following set of key values: 2, 3, 5, 7, 11, 17, 19, 23, 31.

Step 1. Assume that the tree is initially empty and values are added in ascending order. Construct B+ tree for the cases where the number of pointers is as follows:

a) Four b) Six c) Eight

Step 2. For each B+ tree of Step 1, show the form of the tree after each of the following series of operations:

a) Insert 9. b) Insert 10. c) Insert 8.

d) Delete 23. e) Delete 19.

Testing B+ Trees (12)

Exercise 6:

- For a B+ tree with maximum keys = 4, insert the following keys in order: 10, 20, 30, 40, 50, 60, 70, 80, 90.

- Assuming keys increasing by 10, what is the first key added that causes the B+ tree to grow to height 3?
- Show the tree after deleting the following keys: 70, 90, 10.

- In the following, you find three algorithms that might help you to implement the B+ Tree.

```
function containsValue(int nodeID, int value) : boolean
    node ← get the node for nodeID;
    if node is a leaf node then
        if value is found in node then
            return true
        else
            return false
    else // node is a branch node, continue with child
        childID ← child ID with greatest key k so that k ⩽ value in node;
        return containsValue(childID, value)
```

Algorithm 1: Checking if a value is contained in a B+ Tree.

```
function insertValue(int nodeID, int value) : long
    node ← get the node for nodeID;
    if node is a leaf node then
        if node already contains value then
            return NO_CHANGES
        else // value has to be inserted
            increment the number of values stored in the tree;
            if some space is left then
                insert the value into node;
                return NO_CHANGES
            else // node has to be split
                rightID ← create a new leaf node;
                right ← the node with ID rightID;
                distribute values between node and right;
                return keyIDPair(firstValue(right), rightID)
    else // node is a branch node
        childID ← child ID with greatest key k so that k ⩽ value in node;
        result ← insertValue(childID, value);
        if result = NO_CHANGES then // nothing to do
            return NO_CHANGES
        else // child was split
            midKey ← getMidKey(result);
            rightID ← getChildID(result);
            if some space is left in node then
                insert the rightID into node's child ID list;
                insert the midKey into node's value list;
                return NO_CHANGES
            else // node is full and has to be split
                rightID ← create a new branch node;
                right ← the node with ID rightID;
                distribute values and child pointers in node and right;
                midKey' ← middle value between those in node and right;
                return keyIDPair(midKey', rightID)
```

Algorithm 2: Inserting a value into a B+ Tree.

```
function deleteValue(int nodeID, int value) : boolean
    node ← get the node for nodeID;
    if node is a leaf node then
        if value is not found in node then
            return true
        else // value is found
            delete value from node;
            decrement the number of values stored in the tree;
            if node still has enough values stored then
                return true
            else // node is under-full
                return false
    else // node is a branch node
        childID ← child ID with greatest key k so that k ⩽ value in node;
        result ← deleteValue(childID, value);
        if result = true then // nothing to do
            return true
        else // the child has become under-full
            child ← get node with ID childID;
            if child is the only child of node then // node must be the root
                delete the old root node;
                set child as the new root node;
                return true
            else // child has a neighbor below node
                neighbor ← get a neighbor of child;
                if neighbor has more than d values then
                    re-fill child by borrowing from neighbor;
                    return true
                else // neighbor is minimally full
                    merge child with neighbor;
                    adjust the pointers and values of node;
                    delete child;
                    if node still has enough values stored then
                        return true
                    else // node is under-full
                        return false
```

Algorithm 3: Deleting a value from a B+ Tree.

This page [illegible] nally left blank.

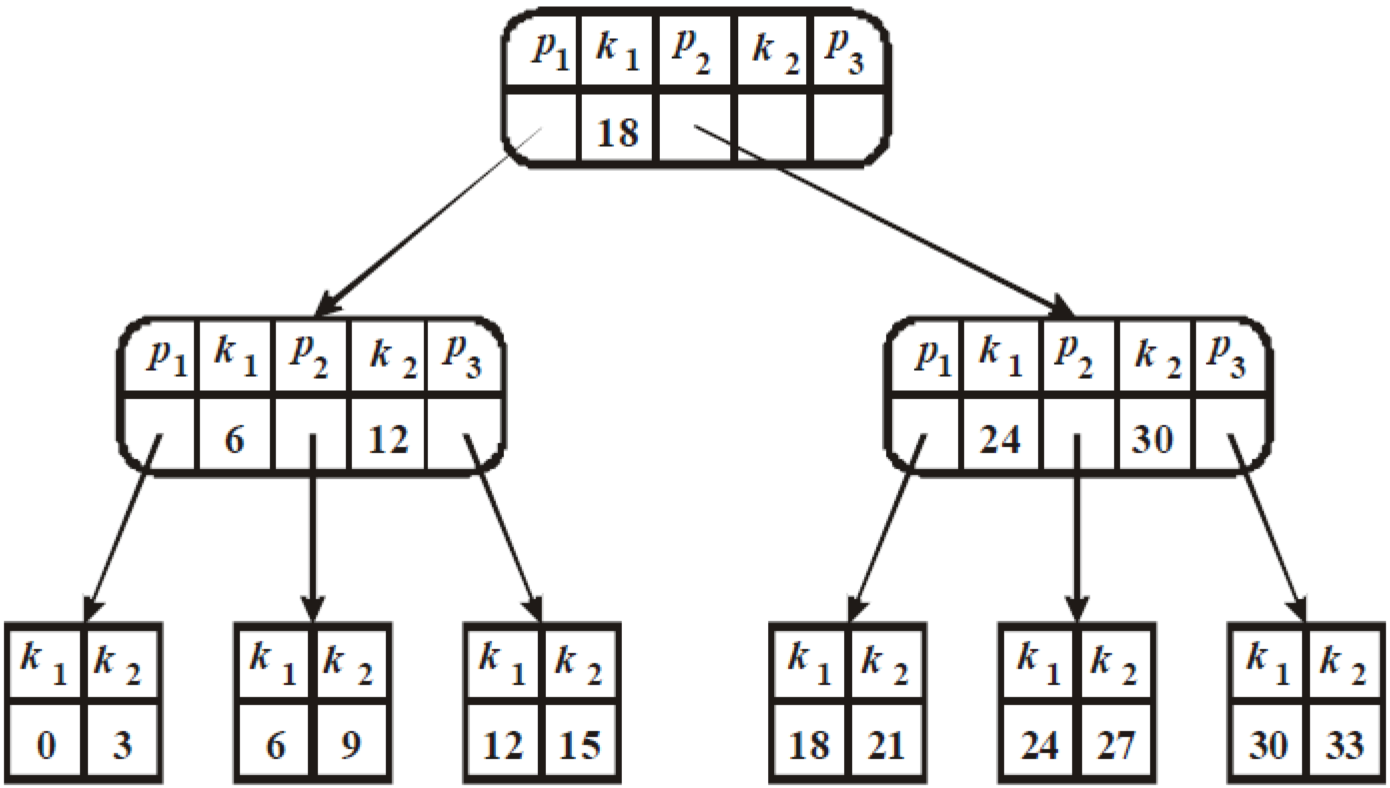
HW #5 B+ Tree

- You will implement a simpler version (**insert** only) of B+ Tree using C / C++ / Java / Python, or **other** programming language if you prefer.
- Your tree will be created and maintained in memory rather than on disk as is commonly the case.

- The task is to write a program that reads integers, store them into a B+ Tree, prints the tree, and prints a sorted list of identifiers.

HW #5 (2)

- A B+ Tree can be represented by listing the nodes using a preorder traversal and indenting them according to their depth.




```
(18,-)
    (6,12)
        [0,3]
        [6,9]
        [12,15]
    (24,30)
        [18,21]
        [24,27]
        [30,33]
```

HW #5 (3)

- The program (BTree) is invoked with command line arguments of variable numbers of program parameters. The first program parameter is the key capacity; then followed by a list of positive integers.
- Sample command lines together with a representative list of integers are shown below:

$ BTree 3 22 16 41 11 18 28 58 1 8 12 17 19 23 31 52 59 61

HW #5 (4)

- Program **BTree** reads an integer key, inserts it into a B+ Tree, and prints a representation of the tree after each integer is inserted. Note that the integer about to be inserted is printed just before the B+ Tree is printed. B+ Trees are printed using a preorder traversal.
- Keys in interior nodes of the B+ Tree are enclosed in parentheses and separated by colons (**:**). Empty positions for keys in a node that is not completely filled are represented by **underscores** (or **dash** if you prefer).
- Nodes in the B+ Tree are indented according to the **depth** of the node. The root is not indented. Each level is indented **five** spaces more than the previous level.
- Following shows the output produced by the program given in the sample list of integers.

```
Insert key 22:
[22,_,_]

Insert key 16:
[16,22,_]

Insert key 41:
[16,22,41]

Insert key 11:
(22:_}
   [11,16,_]
   [22,41,_]

Insert key 18:
(22:_)
   [11,16,18]
   [22,41,_]

Insert key 28:
(22:_)
   [11,16,18]
   [22,28,41]

Insert key 58:
(22:41)
   [11,16,18]
   [22,28,_]
   [41,58,_]

Insert key 1:
(22:_)
   (16:_)
      [1,11,_]
      [16,18,_]
   (41:_)
      [22,28,_]
      [41,58,_]
```

```
Insert key 8:
(22:_)
   (16:_)
      [1,8,11]
      [16,18,_]
   (41:_)
      [22,28,_]
      [41,58,_]

Insert key 12:
(22:_)
   (11:16)
      [1,8,_]
      [11,12,_]
      [16,18,_]
   (41:_)
      [22,28,_]
      [41,58,_]

Insert key 17:
(22:_)
   (11:16)
      [1,8,_]
      [11,12,_]
      [16,17,18]
   (41:_)
      [22,28,_]
      [41,58,_]
```

```
Insert key 19:
(16:22)
   (11:_)
      [1,8,_]
      [11,12,_]
   (18:_)
      [16,17,_]
      [18,19,_]
   (41:_)
      [22,28,_]
      [41,58,_]

Insert key 23:
(16:22)
   (11:_)
      [1,8,_]
      [11,12,_]
   (18:_)
      [16,17,_]
      [18,19,_]
   (41:_)
      [22,23,28]
      [41,58,_]
```

```
Insert key 23:
(16:22)
   (11:_)
      [1,8,_]
      [11,12,_]
   (18:_)
      [16,17,_]
      [18,19,_]
   (41:_)
      [22,23,28]
      [41,58,_]

Insert key 31:
(16:22)
   (11:_)
      [1,8,_]
      [11,12,_]
   (18:_)
      [16,17,_]
      [18,19,_]
   (28:41)
      [22,23,_]
      [28,31,_]
      [41,58,_]

Insert key 52:
(16:22)
   (11:_)
      [1,8,_]
      [11,12,_]
   (18:_)
      [16,17,_]
      [18,19,_]
   (28:41)
      [22,23,_]
      [28,31,_]
      [41,52,58]
```

```
Insert key 59:
(22:_)
   (16:_)
      (11:_)
         [1,8,_]
         [11,12,_]
      (18:_)
         [16,17,_]
         [18,19,_]
   (41:_)
      (28:_)
         [22,23,_]
         [28,31,_]
      (58:_)
         [41,52,_]
         [58,59,_]

Insert key 61:
(22:_)
   (16:_)
      (11:_)
         [1,8,_]
         [11,12,_]
      (18:_)
         [16,17,_]
         [18,19,_]
   (41:_)
      (28:_)
         [22,23,_]
         [28,31,_]
      (58:_)
         [41,52,_]
         [58,59,61]
```

HW #5 (5)

Second test case:

- Following shows with another list of integers with corresponding actions (and outputs):

$ BTree 2 51 29 73 105 15 31

1. Start with an empty B-tree of order 1. Insert key **51**. The resulting tree contains a single leaf.

   [51,-]

2. Insert key **29**. Key 51 is shifted one position to the right preserving the property that keys are always in ascending order.

   [29,51]

3. Insert key **73**. Key 73 is appended to the list of keys in the node. The leaf is now *overfull*.

   [29,51,73]

   3.1. Split the leaf.
   [29,-]
   [51,73]

   3.2. Create an empty interior node that will be the parent of the two leaves.
   (-:-)
       [29,-]
       [51,73]

   3.3. *Copy* the smallest key in the node having the larger keys into the newly created interior node.
   (51:-)
       [29,-]
       [51,73]

   3.4. Assign a pointer to the first leaf **[29,-]** to p_1 and a pointer to the second leaf, **[51,73]** to p_2.

4. Insert key **105**.

(51:-)

[29,-]

[51,73,105]

4.1. The leaf, **[51,73,105]**, is *overfull*.

4.2. Split the leaf

(51:-)

[29,-]

[51, -]

[73,105]

4.3. Create an empty interior node that will be the parent of the two leaves.

(51:-)

[29,-]

(-,-)

[51, -]

[73,105]

4.4. *Copy* the smallest key in the node having the larger keys into the newly created interior node.

(51:-)
[29,-]
(73,-)
[51, -]
[73,105]

4.5. Merge the newly created interior node with the existing interior node.

(51:73)
[29,-]
[51, -]
[73,105]

5. Insert key **15**.

(51:73)
[15,29]
[51, -]
[73,105]

6. Insert key **31**.

(51:73)

[15,29,31]
[51, -]
[73,105]

6.1. The leaf **[15,29,31]**, is *overfull*. Split the leaf into **[15,-]** and **[29,31]**. Hoist the middle key into a new interior node **(29:-)** having pointers to the two leaves **[15,-]** and **[29,31]**.

(51:73)

(29:-)
[15,-]
[29,31]
[51, -]
[73,105]

6.2. The new interior node must be merged with the existing interior node.

(51:73) ? (29:-)

[15,-]
[29,31]
[51, -]
[73,105]

(29:51:73)

[15,-]
[29,31]
[51, -]
[73,105]

6.3. The interior node, **(29:51:73)** is *overfull* and must be split.

(29:51:73)

[15,-]
[29,31]
[51, -]
[73,105]

6.4. Splitting an interior node is different than splitting a leaf. The middle key is *removed* rather than copied. Pointers to the subordinate interior nodes are assigned to pointers, p_1 and p_2 on either side of key **51** in the new root **(51:-)**.

(51:-)
    (29:-)
    (73:-)

6.5. Subordinate leaves remain attached to their respective interior nodes.

(51:-)
    (29:-)
        [15,-]
        [29,31]
    (73:-)
        [51, -]
        [73,105]