B-Trees

Multiway Trees

- For any binary tree (e.g., binary search tree, AVL tree) or even for 2-3 tree, the outdegree is restricted to two or three.
- As the trees grow in size, their height can become significant.
  - A binary tree with 1,000 entries has height of at least 9 while a tree with 100,000 entries has a height of at least 16 assuming that $h = 0$ for single entry.
  - In case of unbalanced trees, the height can be significantly larger.
- For multiway trees, the outdegree is not restricted to two or three while retaining the general properties of binary search trees.
  - A ***m***-way tree is a search tree in which each node can have from zero to ***m*** subtrees, where ***m*** is defined as the order of the tree.
  - A ***m-***way tree is not a balanced tree.
  - A balanced ***m***-way search tree is a B tree.

B-Trees

- A B-tree provides an efficient index organization for data sets of structured records
  - Introduced by R. Bayer and E. McGreight in 1972
  - Extends the idea of the 2-3 tree by permitting more than a single key in the same node of a search tree.
- In a B-tree, all data records (or record keys) are stored at the leaves, in increasing order of the keys
  - The parental nodes are used for indexing
    - Usually called B^+ tree
  - When used for storing a large data file on a disk, the nodes of a B-tree usually correspond to the disk pages (blocks).

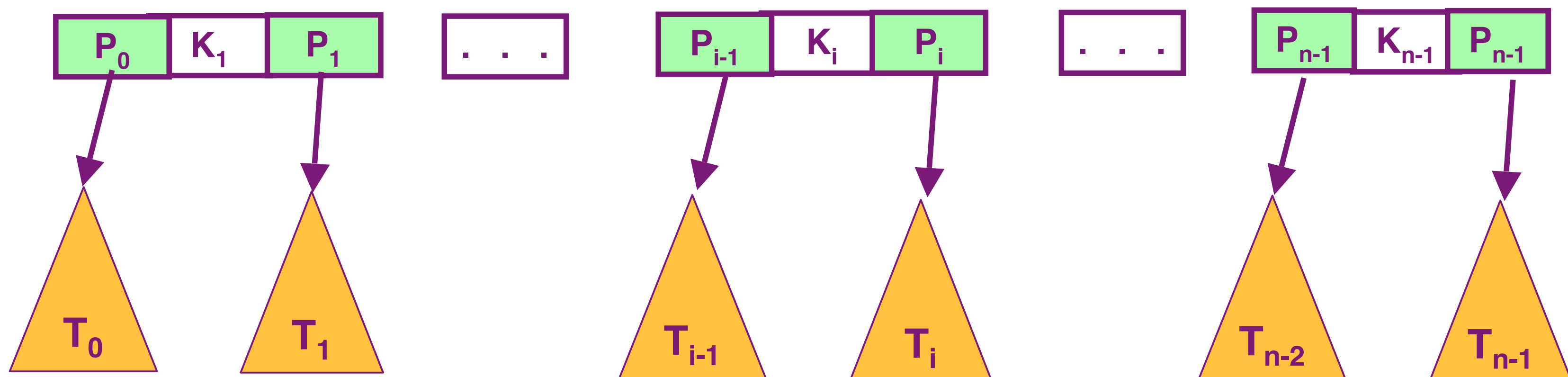
Parental Node of a B-Tree

- Each B-tree node contains ***n* -1** ordered keys $K_1 < K_2 \ldots\ldots < K_{n-1}$

- The keys are interposed with ***n*** pointers or references to the node's children so that all the keys in subtree T_0 are smaller than K_1, all the keys in subtree T_1 are greater than or equal to K_1 and smaller than K_2 with K_1 being equal to the smallest key in T_1, and so on.

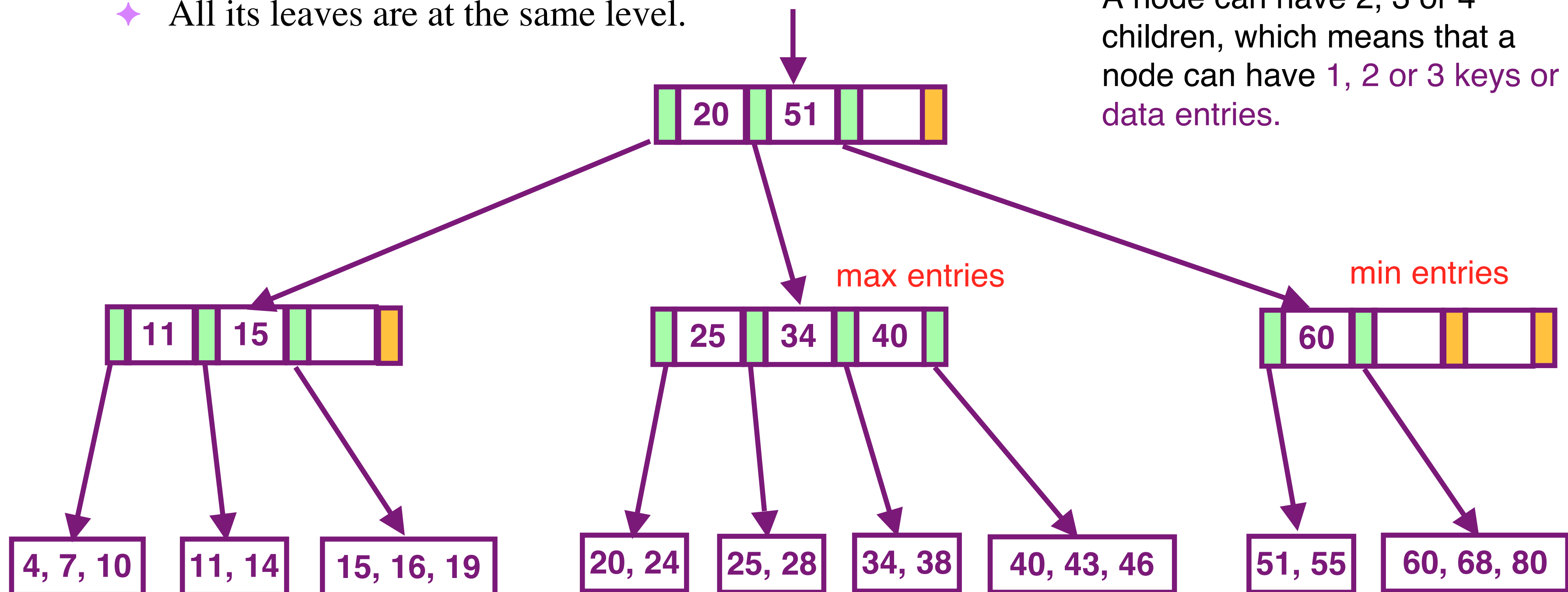
- The keys of the last subtree T_{n-1} are greater than or equal to K_{n-1} with K_{n-1} being equal to the smallest key in T_{n-1}.

- A ***n-node*** is shown here.



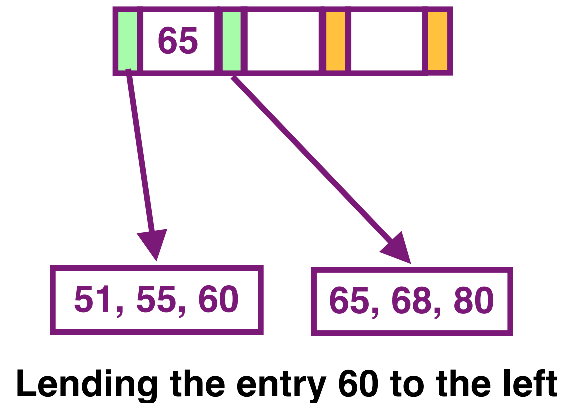
Properties of B-Trees

- The root is either a leaf or has between 2 and ***m*** children.
  - A leaf has between 1 and ***m*** -1 entries or keys.
- Each node, except the root and the leaves, has between ⌈***m***/2⌉ and ***m*** children
  - Has keys between ⌈***m***/2⌉ - 1 and ***m*** -1.
- The tree is perfectly balanced
  - All its leaves are at the same level.



Example of B-Tree Insertion

- Inserting 65 into the B-tree under restriction that the tree's leaves cannot contain more than three items.
- Can also be done by moving 60, the smallest key of the full leaf, to its sibling with keys 51 and 55 and replacing the key value of their parent by 65, the new smallest value in the second child.



Lending the entry 60 to the left

Inserting 65

20 51

11 15

25 34 40

60 68

4, 7, 10
11, 14
15, 16, 19
20, 24
25, 28
34, 38
40, 43, 46
51, 55
60, 65
68, 80

Efficiency of B-Tree

- Time efficiency depends on the height (***h***) of the tree
  - The number of nodes we need to access during a search for a record with a given key value is equal to the height of the tree plus one.

- Find the smallest number of keys a B-tree of order m and height h can have:
  - The root of the tree will contain, at least, one key
  - Level 1 will have at least two nodes with at least $\lceil m/2 \rceil - 1$ keys in each of them, for the total minimum number of keys $2(\lceil m/2 \rceil - 1)$.
  - Level 2 will have at least $2\lceil m/2 \rceil$ nodes (the children of the nodes at level 1) with at least $\lceil m/2 \rceil - 1$ in each of them, for the total minimum number of keys $2\lceil m/2 \rceil(\lceil m/2 \rceil - 1)$.
  - In general, the nodes at the level i, will contain at least $2\lceil m/2 \rceil^{i-1} (\lceil m/2 \rceil - 1)$ keys.
  - Finally, level h, the leaf level, will have at least $2\lceil m/2 \rceil^{h-1}$
  - Thus we have the following inequality

$$n \geq 1 + \sum_{i=1}^{h-1} 2\left\lceil \frac{m}{2} \right\rceil^{i-1} \left(\left\lceil \frac{m}{2} \right\rceil - 1 \right) + 2\left\lceil \frac{m}{2} \right\rceil^{h-1}$$

$$n \geq 4\left\lceil \frac{m}{2} \right\rceil^{h-1} - 1.$$

$$h \leq \left\lfloor \log_{\lceil m/2 \rceil} \frac{n+1}{4} \right\rfloor + 1.$$

The searching in a B-tree is a ***O*(log *n*)** operation

The ***h***'s upper bound is 6, 5 and 4, respectively, for order ***m*** of 50, 100, 250 for a file of 100 million entries.

B-Tree Abstract Data Type

- B-Tree data structure: Implementation requires three types of structures:
  - Head structure (B_TREE)
  - Data node (NODE)
  - Entry structure (ENTRY)

- B-Tree functions to build and maintain the tree:
  - Basic algorithms
    - ✓ BTreeCreate, BTreeInsert, BTreeDelete
  - Tree data processing
    - ✓ BTreeRetrieval, BTreeTraversal
  - Tree utility functions
    - ✓ BTreeSearch, BTreeEmpty, BTreeFull, BTreeCount, BTreeDestroy

```
B_TREE
   count            <integer>
   root             <node pointer>
   compare          <pointer>
end B_TREE

NODE
   firstPtr         <node ponter>
   numEntries       <integer>
   entries[m-1]     <ENTRY>
end NODE

ENTRY
   key              <keyType>
   data             <dataType>
   rightPtr         <node pointer>
end ENTRY
```

Searching

- Similar to searching in the binary search tree and even more so in the 2-3 tree.
- Starting with root, we follow a chain of pointers to the leaf that may contain the search key.
- Search for the search key among the keys of that leaf.
- Since keys are stored in the sorted order, at both parental nodes and leaves, one can use binary search if the number of keys in a node is very large.

Insertion

- Apply search procedure to the new record's key K to find the appropriate leaf for the new record.
- If there is room for the record in that leaf, place it there.
- If there is no room for the record, the leaf is split in half by sending the second half of the records to a new node
  - The smallest key K in the new node and the pointer to it have to be inserted into the old leaf's parent.
- This recursive procedure may percolate up to the tree's root.
  - If the root is already full too, a new root is created with the two halves of the old keys split between two children of the new root.
  - One can avoid the possibility of recursive node splits by
    - ✓ Splitting full nodes as we are searching for an appropriate leaf for the new record.
    - ✓ Moving a key to the node's sibling to avoid some node splits.

Insert logic requires several algorithms

ADT Interface function

Internal functions:

Search node, Insert node, Insert entry and Split node functions

B-Tree Deletion

- Process involves several steps:
  - Search for the data to be deleted.
  - Once the data entry is deleted, determine whether it has caused an underflow.
    - A deletion that results in a node with fewer than minimum number of entries is an underflow.
  - Correct the underflow structural deficiency as node delete backs out of the recursion.

- Delete logic requires two algorithms
  - ADT delete interface
    - This user interface function calls the recursive delete function and checks for root overflow.
  - Internal functions
    - Delete entry
    - Delete middle
    - Reflow

B-Tree Delete Internal Functions

- Delete node: Searches for the node (delete key) to be deleted. If the key is not in the current node, it calls itself recursively with a new subtree. If the key is found, it deletes the key by calling either delete entry or delete mid. Then it checks for underflow and if necessary repairs the underflow (reflow).

- Delete entry: Removes the entry from a node and compresses it (i.e., moves the entries to the right of the deleted key to the left).

- Delete middle: When the data to be deleted are not in leaf node, then the immediate predecessor (i.e., the largest node on the left subtree of the entry to be deleted) is used as the substitute data. The function recursively follows the left subtree's right pointer in the last entry until it comes to a leaf node. Then it replaces the deleted data with the data in the leaf's last entry and then physically deletes the predecessor from the leaf node.

- Reflow: Brings the underflowed node up to a minimum state by adding at least one entry to it.
  - Balance: shifts an entry from one sibling to another through the parent.
    - ✓ Borrow left: Checks whether the left subtree is above the minimum, and if it is, burrows one entry from the left to right subtree which has gone underflowed.
    - ✓ Borrow right: Burrows one entry from the right subtree if it is above minimum to the underflowed left subtree.
  - Combine: Joins the data from an underflowed entry, a minimal sibling and a parent in one node
    - ✓ It results in one node with the maximum number of entries.
    - ✓ The empty node is recycled.

Balance: Borrow from Left

- Steps: Right subtree borrows the left.
  - First shift data right
  - Rotate parent node down
  - Rotate data up

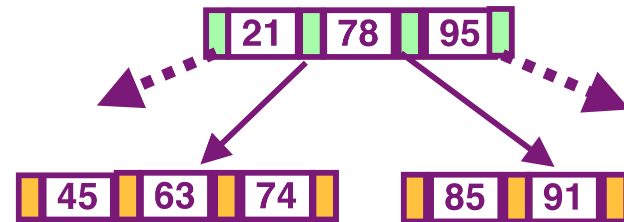
- For order $m = 5$,

  Minimum number of of entries

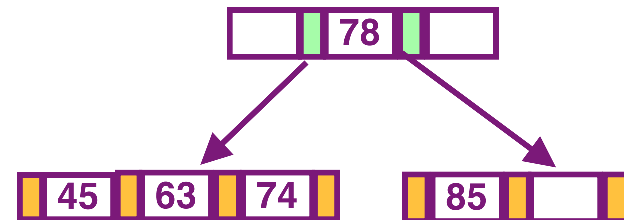
  $= \lceil m/2 \rceil - 1 = 2$

  Maximum number of entries

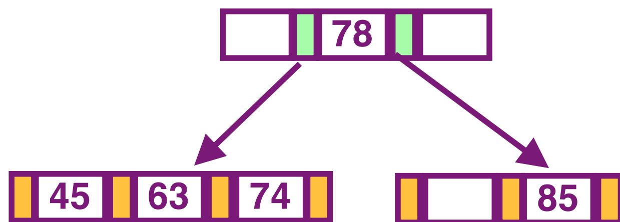
  $= m - 1 = 4$



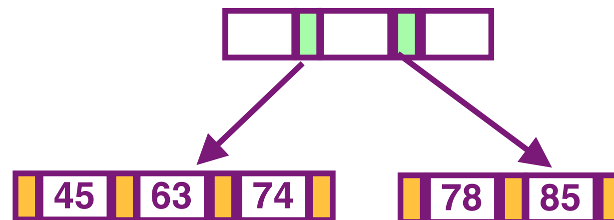
Deleting 91 in the right subtree causes an underflow



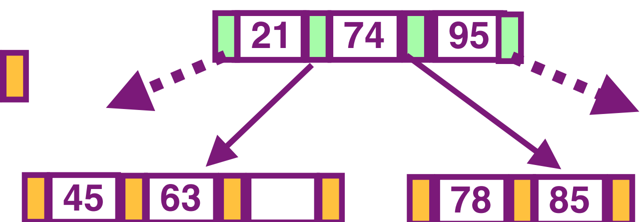
Underflowed node



Right shift of 85



Parent (78) down-rotation



Left data (74) up-rotation

Balancing by by borrow left algorithm

Combine

- Steps: Combines the underflowed node (right node, 78), the left subtree (with minimal entries, 45 and 63) and their parent node 74.
  - Move root to the left subtree
  - Move entries in the underflowed node to the left subtree
  - Shift the root.

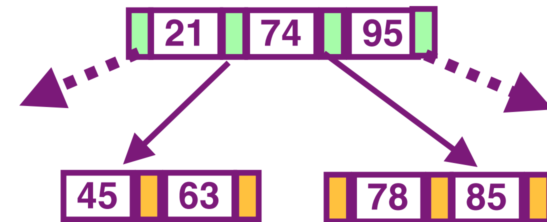
- For order $m = 5$,

  Minimum number of of entries

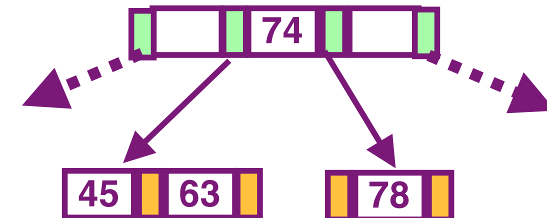
  $= \lceil m/2 \rceil - 1 = 2$

  Maximum number of entries

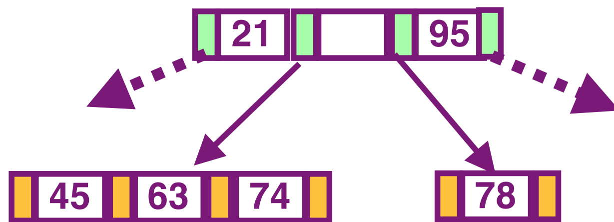
  $= m - 1 = 4$



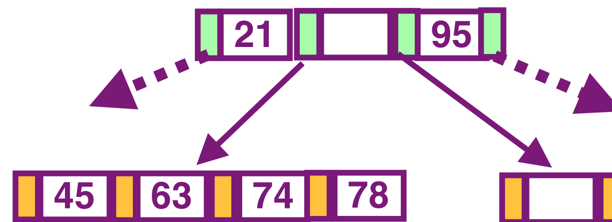
Deleting 85 in the right subtree causes an underflow



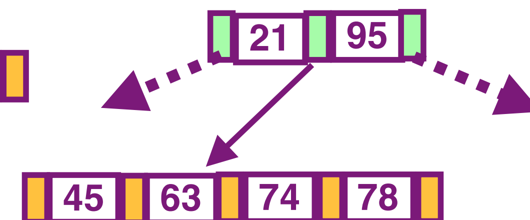
Underflowed node



Move parent 74 to left



Move right entry 78



Shift root

Balancing by by burrow left algorithm