- B+ Tree Index Variations

Supported Search Operations

Tree structured indexing techniques support both range and equality search.
Equality Search: e.g., find the student with `sid="111222"`
Range Search: e.g., find all students with $gpa > 3$

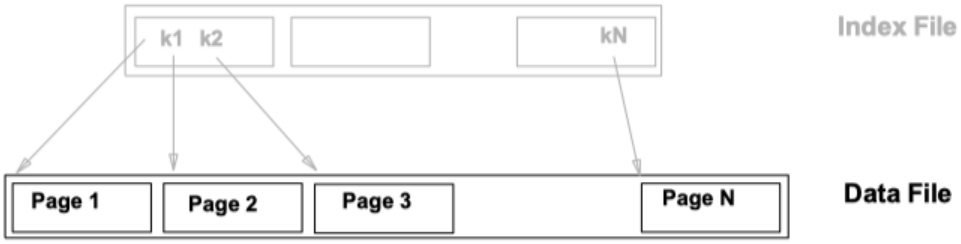
Range Searches

Motivation

If data was stored in a sorted file

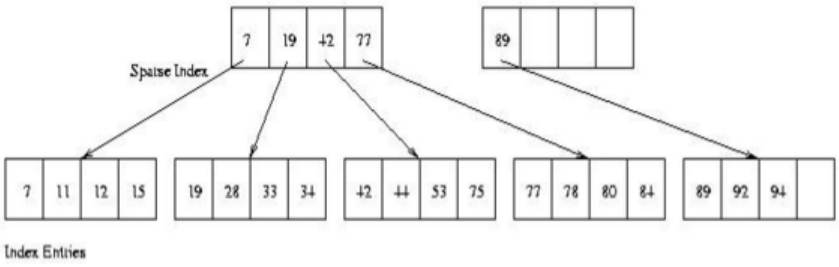
- Can use **binary search** to find first such student, then scan the file from that point to find others
- Cost: log_2F (can be quite high for a large file)
- Can we reduce the cost?

Simple Idea: Create an Index File



1. Use binary search (index entries sorted)
2. If Q pages if index entries
   - then log_2Q page transfers is a big improvement over binary search of an F page data file since $F >> Q$
3. Binary search of the index file could still be expensive for inserts and deletes
4. **Use multilevel index (trees)** : Sparse index on sorted list of index entries
5. Repeated construction of a one-level index leads to a tree structure with several levels of non-leaf pages

Two-Level Index



Non-leaf pages

Contain separators

Separators

Sparse index over pages of index entries

Leaf Pages

Cost of retrieving row once index entry is found is 0 (if integrated) or 1 (if not)

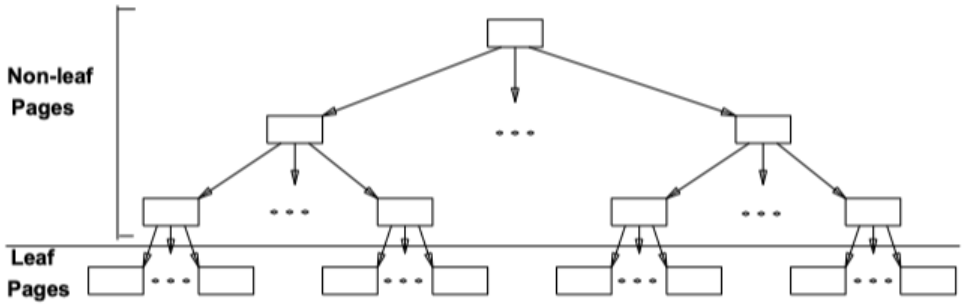
Interpretation 1

Leaf pages contain **index entries** with pointers with data records in a separate data file

Interpretation 2

Leaf pages contain **data records** (clustered index)

Multilevel Index



Cost is

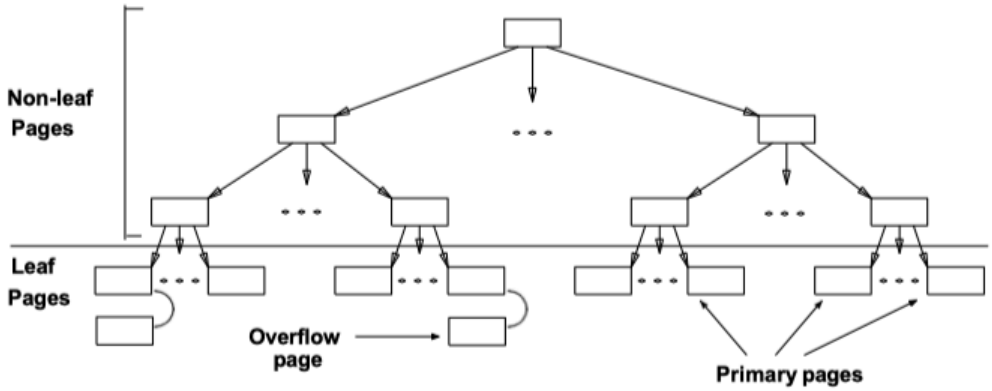
$$log_F Q + 1$$

- F : fanout of a separator page - number of child nodes that each node in the tree can have
- Q: Total number of keys
  As a consequence

$$\text{cost} = \text{no. of levels in the tree}$$

  Example: If
- F = 100
- Q = 10 000
- cost = 3 (reduced to 2 if root is kept in main memory)

Index Sequential Access Method (ISAM)



ISAM is a **static index structure**

File creation

1. Leaf (data) pages allocated sequentially
2. Sorted by search key
3. Index pages allocated
4. Space for overflow pages allocated

Index Entries

- they direct the search for data entries, which are in leaf pages

Search

Cost is

$$log_F N$$

- F : fanout of a separator page - number of child nodes that each node in the tree can have
- Q: number of leaf pages

Insert and Deletes only affect leaf pages

Insert

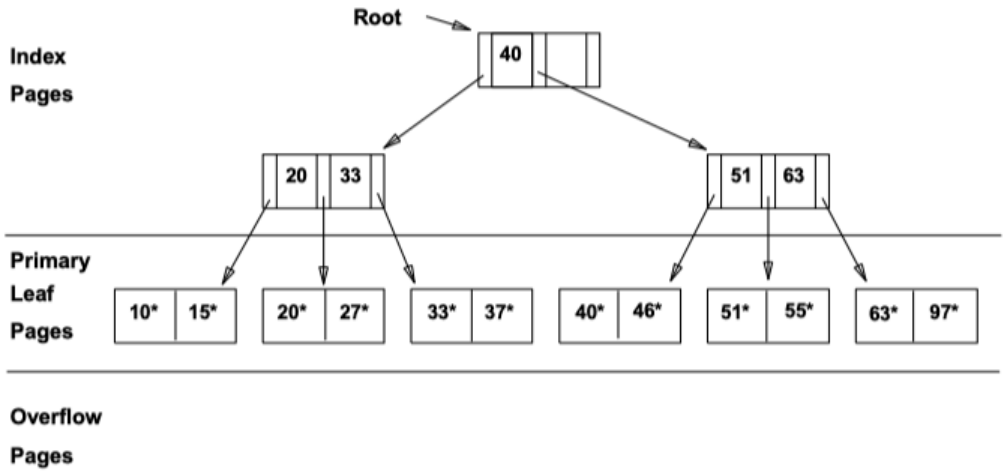
1. Find the leaf page data entry belongs to
2. Place it there
3. If there is no space, allocate an overflow page

Delete

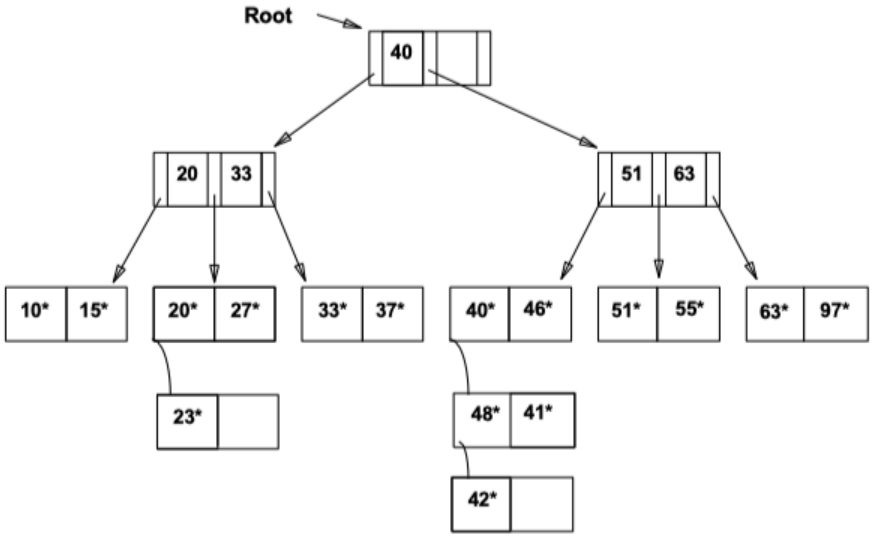
1. Find and remove from leaf
2. If empty overflow page, de-allocate

Example

Suppose each node can hold 2 entries

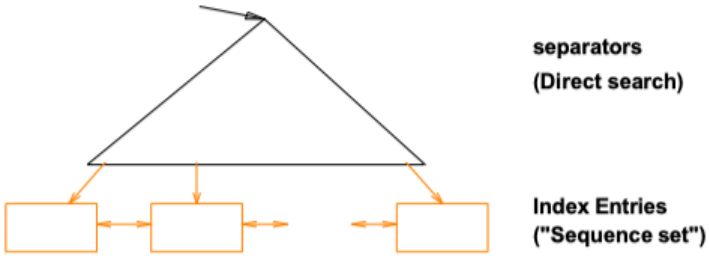


Inserting 23, 48, 41, 42



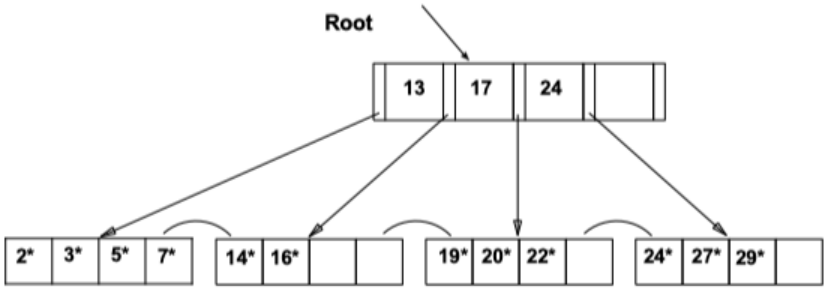
B+ Tree

Main Features



1. Search/insert/delete guaranteed at $log_F N$ ($F = \text{fanout}$, $N = \text{leaf pages}$)
2. Except for the root, the minimum occupancy is 50%, $\phi/2$ (*where ϕ is the maximum capacity of a node*)
3. Leaf pages form a sequence set
4. Everything else is much like ISAM

Search



Example: Search for 5, *15* and all data entries >= 20*

Searching for 5*

We follow the left-most pointer since 5 > 13

Search for entry 15*

We follow the second pointer since $13 < 15 < 17$

Search for entry 24*

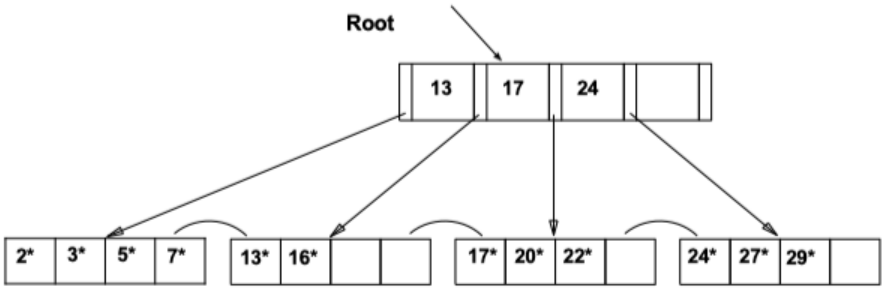
We follow the fourth child pointer since $24 \leq 24 < 30$

Insert

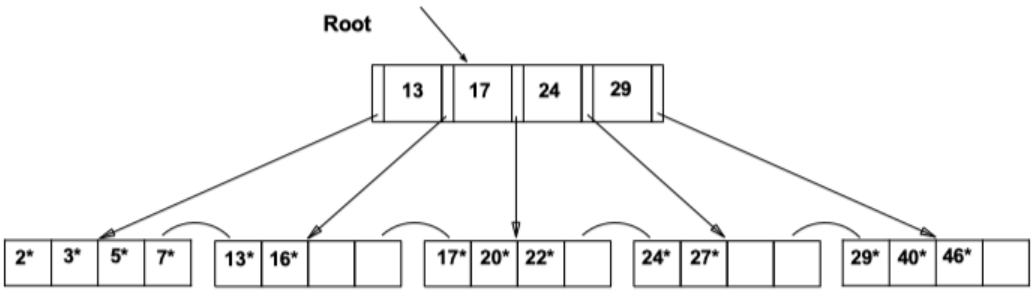
Process

1. Find the correct leaf L
2. Put data entry onto L
   - If L has enough space, DONE!
   - Otherwise, must split L I (into L and a new node L2)
     - Redistribute entries evenly, **copy up** middle key
     - Insert index entry pointing L2 into parent key
3. Splits "grow" tree; root split increases height
4. **Tree growth**: gets wider or one level taller at top

Example

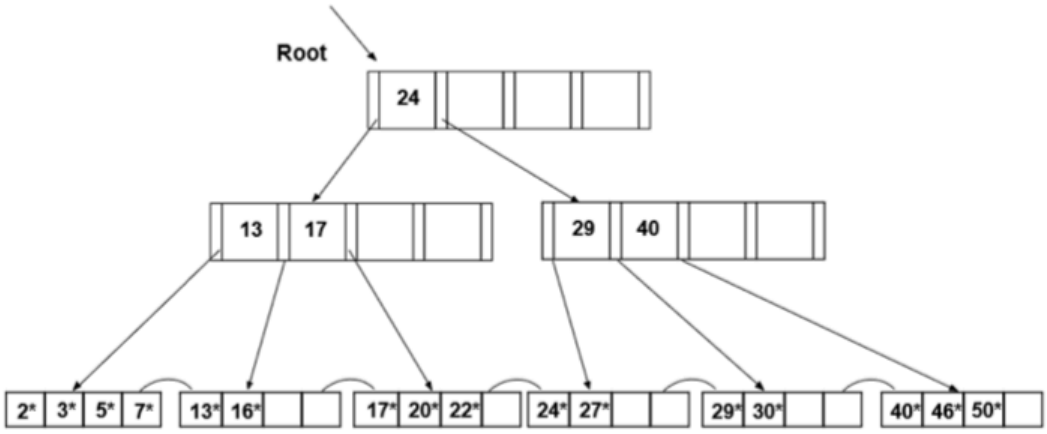


Say we insert 40 *and 46*



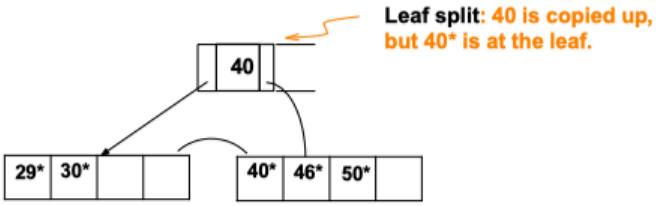
1. Once we add 40, we initially add it on the 4th pointer - occupying the last spot of that pointer. Thus the pointer to the right of 29 points to `[24, 27, 29, 40]`
2. When we attempt to add 46, it causes a leaf split, so we copy up
   - We split it and pushes up 29 as a key, and then add 46

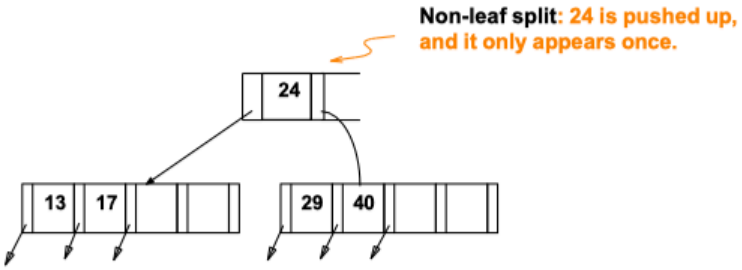
Now we insert 50 *and 30*



- split propagates to the root
- non-leaf split: push up

Split policy

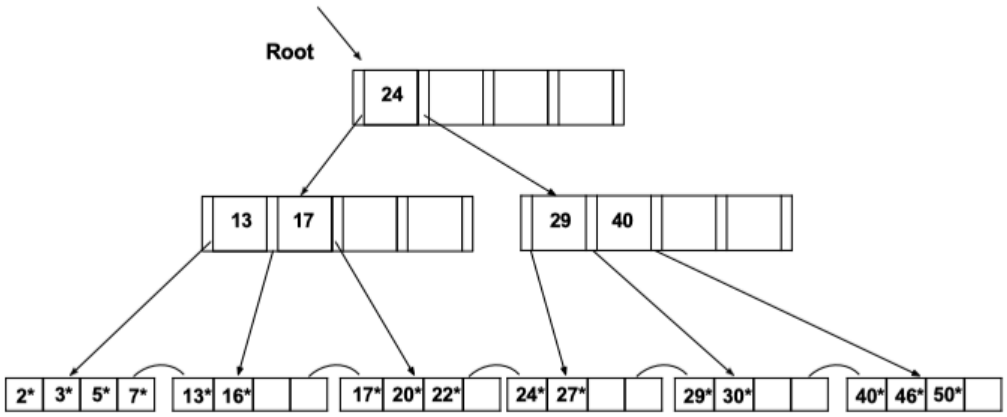




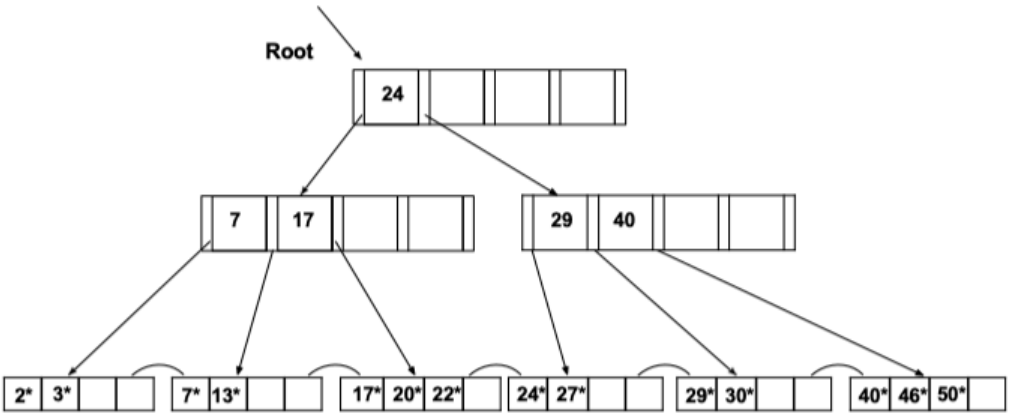
Deleting a data entry from a B+ Tree

1. Find the correct leaf node
2. Remove the entry from the node
3. If the node is at least half full, done!
4. Else possibly borrow some entries from a sibiling
5. If not possible, merge the node with the sibling
6. Delete the separator between the node and the sibling from the parent node
7. Go to step 3

Example

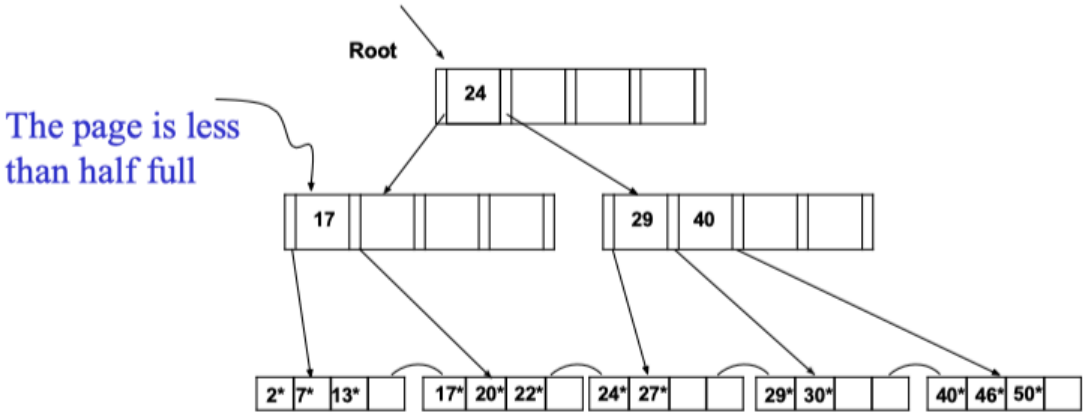


Delete 5 *and 16*

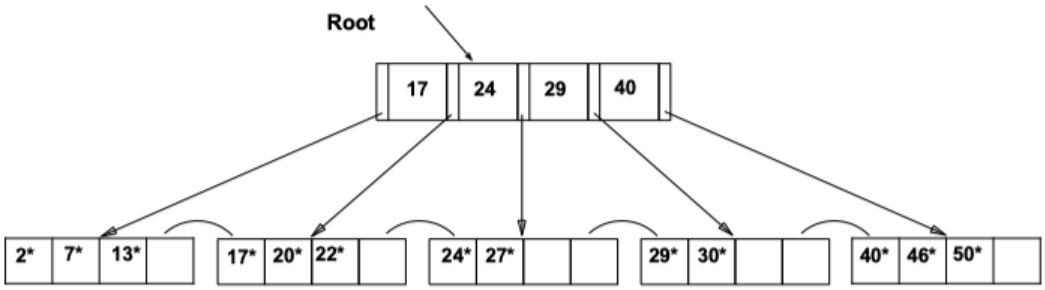


1. Deleting 5 - no problem
2. Deleting 6
    - the page becomes less than half full!
    - Borrow some keys from a neighbour (redistribute the keys equally between them): **copy up**

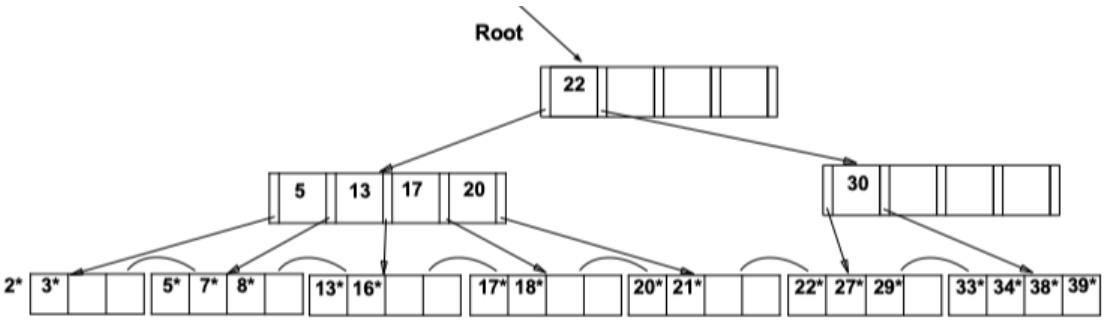
Delete 3*



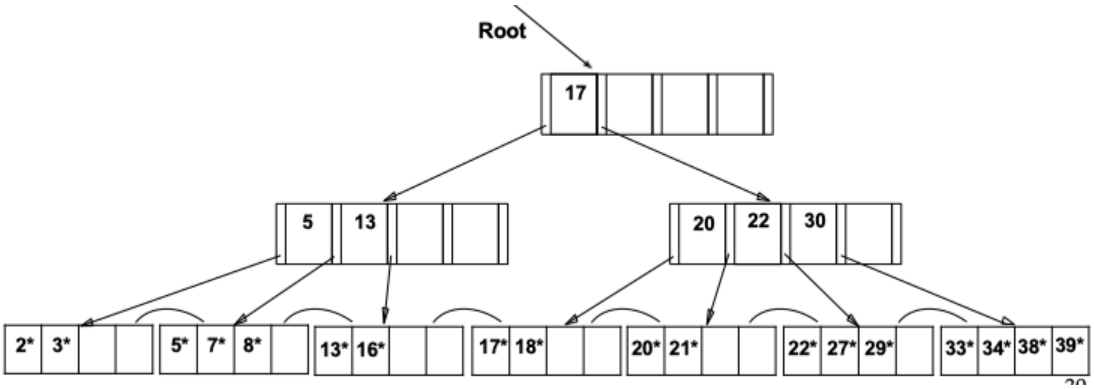
But we cannot borrow from neighbour! So we merge again!



Another Example of Delete



- The node in the middle layer is less than half full
- Redistribute the keys between the page and its neighbour



- Intuitively, entries are **re-distributed by pushing through** the splitting entry in the parent node
- It suffices to re-distribute index entry with key 20; we have re-distributed 17 as well for illustration

B+ Trees Rules for Candidate Selection

Insertion

Select the median key to push up (if even, choose the **left** middle)

Deletion

Borrow

1. Borrowing from **left sibling**: pull the **right most key**
2. Borrowing from **right sibling**: pull the **left most key**

Merge

Pull the **parent key** between the two nodes in into the merged node

B+ Trees in Practice

Typical Trees

- Maximum fanout: 200
- fill-factor: 67%
- average fanout: 133

Typical Capacities

- Height 4: $133^4 = 312\,900\,700$ index entries
- Height 3: $133^3 = 2\,352\,637$ index entries

Can often hold top levels in buffer pool

- Level 1 = 1 page = 8 KB
- Level 2 = 133 pages = 1 MB
- Level 3 = 17 689 pages = 133 MB

B+ Tree Index Variations

- Index entry
  - `<full record>`
  - `<key, address(es)>`
  - `<key, address(es), some other columns>`
- Character string keys
- Variable length keys
- Prefix B+-tree