Multiway Search Tree (MST)

- Generalization of BSTs
- Suitable for disk
- MST of order $n$:
  - Each node has $n$ or fewer sub-trees
    - $S_1 \ S_2 \ldots \ S_m$, $m \leq n$
  - Each node has $n-1$ or fewer keys
  - $K_1 \ K_2 \ldots \ K_{m-1}$: $m-1$ keys in ascending order
    - $K(S_i) \leq K_i \leq K(S_{i+1})$, $K(S_{m-1}) < K(S_m)$
MSTs for Disk

- Nodes correspond to disk pages
- **Pros:**
  - tree height is low for large \( n \)
  - fewer disk accesses
- **Cons:**
  - low space utilization if non-full
  - MSTs are non-balanced in general!
4000 keys, n=5

- At least $4000/(5-1)$ nodes (pages)
  - $1^{st}$ level (root): 1 node, 4 keys, 5 sub-trees +
  - $2^{nd}$ level: 5 nodes, 20 keys, 25 sub-trees +
  - $3^{rd}$ level: 25 nodes, 100 keys, 125 sub-trees +
  - $4^{th}$ level: 125 nodes, 500 keys, 525 sub-trees +
  - $5^{th}$ level: 525 nodes, 2100 keys, 2625 sub-trees +
  - $6^{th}$ level: 2625 nodes, 10500 keys, ...
- tree height = 6 (including root)

- If n = 11 at least 400 nodes and tree height = 3
Operations on MSTs

- **Search**: returns pointer to node containing the key and position of key in the node
- **Insert**: new key if not already in the tree
- **Delete**: existing key
Important Issues

- Keep MST balanced after insertions or deletions
- Balanced MSTs: B-trees, B+-trees
- Reduce number of disk accesses
- Data storage: two alternatives
  1. inside nodes: less sub-trees, nodes
  2. pointers from the nodes to data pages
Search Algorithm

\textbf{MST *search} (MST *tree, int n, keytype key, int *position) {
    MST *p = tree;
    while (p != null) { // search a node
        i = nodesearch(p, key);
        if (i < numtrees(p) - 1 && key == key(p, i)) {
            *position = i; return p;  }
        p = son(p, i);
    }
    position = -1;
    return(-1);
}
key = 63
search = C
position = 2
found = true
(a)

key = 137
search = O
position = 3
found = true
(b)

key = 71
search = I
position = 2
found = false
(c)

key = 102
search = N
position = 1
found = false
(d)

key = 22
search = G
position = 4
found = false
(e)

key = 148
search = Q
position = 2
found = false
(f)
Insertions

- Search & insert key if not there
  a) the tree grows at the leafs \( \Rightarrow \) the tree becomes imbalanced \( \Rightarrow \) not equal number of disk accesses for every key
  b) the shape of the tree depends on the insertion sequence
  c) low storage utilization, many non-full nodes
**insert**: 70 75 82 77 71 73 84 86 87 85

$$n = 4$$

max 3 keys/node
MST Deletions

- Find and delete a key from a node
  - free the node if empty
- If the key has right or/and left sub-trees => find its successor or predecessor
  - min value of right subtree or max value of left subtree
- put this in the position of the key and delete the successor or predecessor
- The tree becomes imbalanced
n=4

delete 150, 180
Definition of n-Order B-Tree

- Every path from the root to a leaf has the same length $h \geq 0$
- Each node except the root and the leaves has at least $\left\lfloor \frac{n}{2} \right\rfloor$ sub-trees and $\left\lfloor \frac{n}{2} \right\rfloor - 1$ keys
- Each node has at most $n$ sub-trees and $n-1$ keys
- The root has at least 1 node and 2 sub-trees or it is a leaf
B-Tree Insertions

- Find leaf to insert new node
- If not full, insert key in proper position **else**
- Create a new node and split contents of old node
  - left and right node
  - the \( n/2 \) lower keys go into the left node
  - the \( n/2 \) larger keys go into the right node
  - the separator key goes up to the father node
- If \( n \) is even: one more key in left (left bias) or right node (right bias)
(a) Initial portion of a B-tree

(b) After inserting 382

(c) After inserting 518 and 508
(a) An initial B-tree twig

(b) Inserting 102 with a left bias

(c) Inserting 102 with a right bias

\[ n=4 \]
B-Tree Insertions (cont.)

- If the father node is full: split it and proceed the same way, the new father may split again if it is full ...
- The splits may reach the root which may also split creating a new root
- The changes are propagated from the leafs to the root
- The tree remains balanced
n=5

(b) After inserting 356

(c) After inserting 462

E.G.M. Petrakis
B-trees
Advantages

- The tree remains balanced!!
- Good storage utilization: about 50%
- Low tree height, equal for all leaves
  \[\Rightarrow\] same number of disk accesses for all leaves
- **B-trees**: MSTs with additional properties
  - special insertion & deletion algorithms
B-Tree Deletions

Two cases:

a) **Merging**: If less than \( n/2 \) keys in left and right node then, sum of keys + separator key in father node < \( n \)
   - the two nodes are merged,
   - one node is freed

b) **Borrowing**: if after deletion the node has less than \( n/2 \) keys and its left (or right) node has more than \( n/2 \) keys
   - find successor of separator key
   - put separator in the place of the deleted key
   - put the successor as separator key
(a) Deleting key 113

(b) Deleting key 120 and consolidating

\[ n = 4 \]

\[ n = 5 \]
\begin{align*}
\text{Figure 8.3.10} \\
(a) \text{Deleting 65, consolidating and borrowing} \\
(b) \text{Deleting 173 and a double consolidation}
\end{align*}
Performance

<table>
<thead>
<tr>
<th>n</th>
<th>N</th>
<th>10^3</th>
<th>10^4</th>
<th>10^5</th>
<th>10^6</th>
<th>10^7</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

\[ \log_n (N+1) \leq h \leq \log_{(n+1)/2} ((N+1)/2) + 1 \]

- Search: the cost increases with the \( \log_n N \)
- Tree height \( \sim \log_n N \)
Performance (cont.)

- Insertions/Deletions: Cost proportional to $2h$
  - the changes propagate towards the root
- Observation: The cost drops with $n$
  - larger size of track (page) but, in fact the cost increases with the amount of information which is transferred from the disk into the main memory
Problem

- B-trees are good for **random** accesses
  - E.g., search(30)
- But not for **range** queries: require multiple tree traversals
  - search(20-30)
B+-Trees

- At most $n$ sub-trees and $n-1$ keys
- At least $\left\lfloor \frac{n}{2} \right\rfloor$ sub-trees and $\left\lfloor \frac{n}{2} \right\rfloor - 1$ keys
- Root: at least 2 sub-trees and 1 key
- The keys can be repeated in non-leaf nodes
- Only the leafs point to data pages
- The leafs are linked together with pointers
index

data pages
Performance

- Better than B-trees for range searching
- B-trees are better for random accesses
- The search must reach a leaf before it is confirmed
  - internal keys may not correspond to actual record information (can be separator keys only)
  - insertions: leave middle key in father node
  - deletions: do not always delete key from internal node (if it is a separator key)
Insertions

- Find appropriate leaf node to insert key
- If it is full:
  - allocate new node
  - split its contents and
  - insert separator key in father node
- If the father is full
  - allocate new node and split the same way
  - continue upwards if necessary
  - if the root is split create new root with two sub-trees
Deletions

- Find and delete key from the leaf
- If the leaf has < n/2 keys
  a) borrowing if its neighbor leaf has more than n/2 keys update father node (the separator key may change) or
  b) merging with neighbor if both have < n keys
     - causes deletion of separator in father node
     - update father node
- Continue upwards if father node is not the root and has less than n/2 keys
data pages: min 2, max 3 records
index pages: min 2, max 3 pointers
B+ order: n = 4, n/2 = 2

data pages don’t store pointers as non-leaf nodes do but they are kinked together
insert 55 with page splitting

delete 84 with borrowing:
delete 31 with merging

<table>
<thead>
<tr>
<th>19</th>
<th>46</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>46</td>
</tr>
</tbody>
</table>

free page
B-trees and Variants

- Many variants:
  - **B-trees**: Bayer and Mc Greight, 1971
  - **B+-trees, B*-trees** the most successful variants

- The most successful data organization for secondary storage (based on primary key)
  - very good performance, comparable to hashing
  - fully dynamic behavior
  - good space utilization (69% on the average)
B+/B-Trees Comparison

- **B-trees**: no key repetition, better for random accesses (do not always reach a leaf), data pages on any node

- **B+-trees**: key repetition, data page on leaf nodes only, better for range queries, easier implementation