CS127: B-Trees
Data Layout on Disk

- Track: one ring
- Sector: one pie-shaped piece.
- Block: intersection of a track and a sector.
Disk Based Dictionary Structures

Use a disk-based method when the dictionary is too big to fit in RAM at once.

Minimize the expected or worst-case number of disk accesses for the essential operations (put, get, remove).

Keep space requirements reasonable -- $O(n)$.

Methods based on binary trees, such as AVL search trees, are not optimal for disk-based representations. The number of disk accesses can be greatly reduced by using $m$-way search trees.
Indexed Sequential Access Method (ISAM)

Store \( m \) records in each disk block.

Use an index that consists of an array with one element for each disk block, holding a copy of the largest key that occurs in that block.
ISAM Limitations

Problems with ISAM:

What if the index itself is too large to fit entirely in RAM at the same time?

Insertion and deletion could be very expensive if all records after the inserted or deleted one have to shift up or down, crossing block boundaries.
Motivation for B-Trees

• Index structures for large datasets cannot be stored in main memory
• Storing it on disk requires different approach to efficiency
• A B-tree is an excellent data structure for storing huge amounts of data for fast retrieval.
Definition of a B-tree

- A B-tree of order $m$ is an $m$-way tree (i.e., a tree where each node may have up to $m$ children) in which:
  1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
  2. all leaves are on the same level
  3. all non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children
  4. the root is either a leaf node, or it has from two to $m$ children
  5. a leaf node contains no more than $m - 1$ keys
B-Tree Example with \( m = 5 \)

Root has at least 2 children.

Each non-root internal node has between \( \lceil m/2 \rceil \) and \( m \) children.

All leaf nodes are at the same level. (External nodes are actually represented by null pointers in implementations.)
We find the location for 10 by following a path from the root using the stored key values to guide the search.

The search falls out the tree at the 4th child of the 1st child of the root.

The 1st child of the root has room for the new element, so we store it there.
We fall out of the tree at the child to the right of key 10.
But there is no more room in the left child of the root to hold 11.
Therefore, we must split this node...
The $m + 1$ children are divided evenly between the old and new nodes.

The parent gets one new child. (If the parent become overfull, then it, too, will have to be split).
Removing 8 might force us to move another key up from one of the children. It could either be the 3 from the 1st child or the 10 from the second child.

However, either of child has no more than the minimum number of children (2), so the two nodes will have to be merged. Nothing moves up.
The root contains one fewer key, and has one fewer child.
Removing 13 would cause the node containing it to become under m/2 children.

To fix this, we try to reassign one key from a sibling that has spares.
The 13 is replaced by the parent’s key 12.
The parent’s key 12 is replaced by the spare key 11 from the left sibling.
The sibling has one fewer element.
11 is in a non-leaf, so replace it by the value immediately preceding: 10.
10 is at leaf, and this node has 3 nodes currently, so just delete it there.
Remove 11 (Cont)
Although 2 is at leaf level, removing it leads to an underfull node.

The node has no left sibling. It does have a right sibling, but that node is at its minimum occupancy already.

Therefore, the node must be merged with its right sibling.
The result is illegal, because the root does not have at least 2 children. Therefore, we must remove the root, making its child the new root.
The new B-tree has only one node, the root.
Let’s put an element into this B-tree.
Adding this key make the node overfull, so it must be split into two. But this node was the root. So we must construct a new root, and make these its children.
The middle key (12) is moved up into the root.
The result is a B-tree with one more level.
Conclusion: B-Trees
Inserting into a B-Tree

- Attempt to insert the new key into a leaf
- If this would result in that leaf becoming too big, split the leaf into two, promoting the middle key to the leaf’s parent
- If this would result in the parent becoming too big, split the parent into two, promoting the middle key
- This strategy might have to be repeated all the way to the top
- If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher
Removal from a B-tree

- During insertion, the key always goes *into a leaf*. For deletion we wish to remove *from a leaf*. There are three possible ways we can do this:
  - 1 - If the key is already in a leaf node, and removing it doesn’t cause that leaf node to have too few keys, then simply remove the key to be deleted.
  - 2 - If the key is *not* in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf -- in this case we can delete the key and promote the predecessor or successor key to the non-leaf deleted key’s position.
Removal from a B-tree (2)

- If (1) or (2) lead to a leaf node containing less than the minimum number of keys then we have to look at the siblings immediately adjacent to the leaf in question:
  - 3: if one of them has more than the min. number of keys then we can promote one of its keys to the parent and take the parent key into our lacking leaf
  - 4: if neither of them has more than the min. number of keys then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leave the parent with too few keys then we repeat the process up to the root itself, if required
B+ -Trees
B+-Tree

- A B+ tree is a data structure often used in the implementation of database indexes. Each node of the tree contains an ordered list of keys and pointers to lower level nodes in the tree. These pointers can be thought of as being between each of the keys. finding an available space
Inserting into a B+-Tree

- If the node has an empty space, insert the key/reference pair into the node.
- If the node is already full, split it into two nodes, distributing the keys evenly between the two nodes. Move the middle child to its parent, if the parents node is full, keep split up until finding an available space.
Removal from a B+ -tree

• Remove the required key and associated reference from the node.

• If the node still has enough keys and references to satisfy the invariants, stop.

• If the node has too few keys to satisfy the invariants, but its next oldest or next youngest sibling at the same level has more than necessary, distribute the keys between this node and the neighbor. If the next oldest or next youngest sibling is at the minimum for the invariant, then merge the node with its sibling.
Root has at least 2 children.

Each non-root internal node has between $\lceil m/2 \rceil$ and $m$ children.

All leaf nodes are at the same level and use pointer to point to the next leaf node. (External nodes are actually represented by null pointers in implementations.)
There is a space in the leftmost leaf node so that we can insert 10 directly.
Insert 10 (Cont)
Since the number of children in the leftmost leaf node will be over the full capacity, it needs to split up and move 8 upwards.
Insert 11 (Cont)
Since the number of children in the leftmost leaf node is less than \( \frac{m}{2} \), we need to merge its neighbor, and move 8 downwards as well.
Remove 2 (Cont)
Let $h =$ height of the B-tree.

get(k): $O(h)$

put(k): $O(h)$

remove(k): $O(h)$. 
B Tree VS B+ Tree

B+ tree also has linked value pairs at the leaf level making it easier to do range queries. B-tree is better for searches of key, value pairs that are closest to the root. B+-tree is more memory efficient on queries, because you only have to read in keys as you traverse the tree, as opposed to keys and tuples in a B-tree.

However, B+-trees use extra insertion and deletion overhead, space overhead. But the advantages of B+-trees outweigh disadvantages, and they are used extensively.
Construct a B+-tree with the pointer of pointer 4 for the following set of key values: (4, 5, 6, 7, 10, 12).
Practice

Insert 4,5,6

Insert 7
Practice

Insert 10

Insert 12
Thank you!
Have a nice day!