Algorithms and Data Structures (CSci 115)

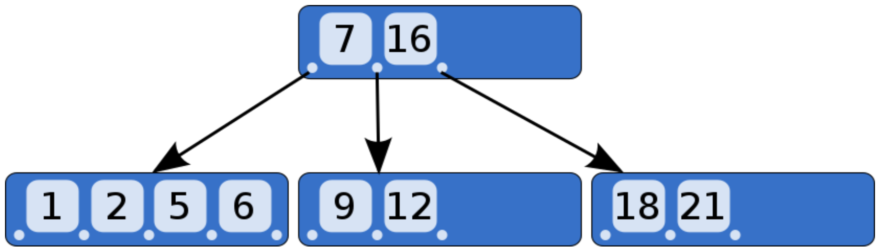
California State University Fresno
College of Science and Mathematics
Department of Computer Science
H. Cecotti

Learning outcomes

- More trees
  - B-trees (1971)
    - Definition
    - Search, Insert, Delete an element
- You must know how to implement and use these trees.

B-tree - Definitions

- Generalization of the 2-3 tree
  - 2-3 tree: B-tree of order **3**.
    - Order = Number of children
- B-tree (1971)
  - Applications
    - Databases
    - Disk-based storage systems
  - Features
    1. keep keys in sorted order for sequential traversing
    2. use a hierarchical index to minimize the number of disk reads
    3. use partially full blocks to speed insertions and deletions
    4. keep the index balanced with a recursive algorithm



B-tree - Definitions

- **Definitions**
  - All leaves are at same level.
  - A B-Tree is defined by the term **minimum degree t**
    - t depends on the “disk block size”.
  - **Every node except root** must contain **at least**: **t-1 keys**.
    - Root may contain minimum 1 key.
  - All the nodes (with the root) may contain at most: 2t – 1 keys.
    - A node is ***full*** if it contains exactly 2t-1 keys
  - Number of children of a node = the number of keys in it **+1**.
  - All the keys of a node are sorted in **increasing** order.
    - The child between 2 keys k1 and k2 contains all keys in range]k1..k2[.
  - B-Tree grows and shrinks **from root**
    - unlike Binary Search Tree!!
    - BSTs grow **downward** and they shrink from downward.
  - Search, insert and delete: O(log n)
    - Like other balanced BSTs

B-tree – 2 Definitions: Degree vs. Order

- **Definitions**
  - Knuth's definition
    - A B-tree of **order** *m* is a tree which satisfies the following properties:

    1. Every node has at most *m* children.
    2. Every non-leaf node (except the root) has at least $\lceil m/2 \rceil$ child nodes.
    3. The root has at least 2 children if it is not a leaf node.
    4. A non-leaf node with *k* children contains $k - 1$ keys.
    5. All leaves appear in the same level and carry no information.
  - → 2-3 Trees: order 3
    - "a B-tree of order 3 is a 2-3 tree"

- **Warning**
  - Different definitions and terminologies in the literature!
    - Order: 2-3 Trees of order 3
    - Degree: 2-3 Trees of degree 2

B-tree - Height

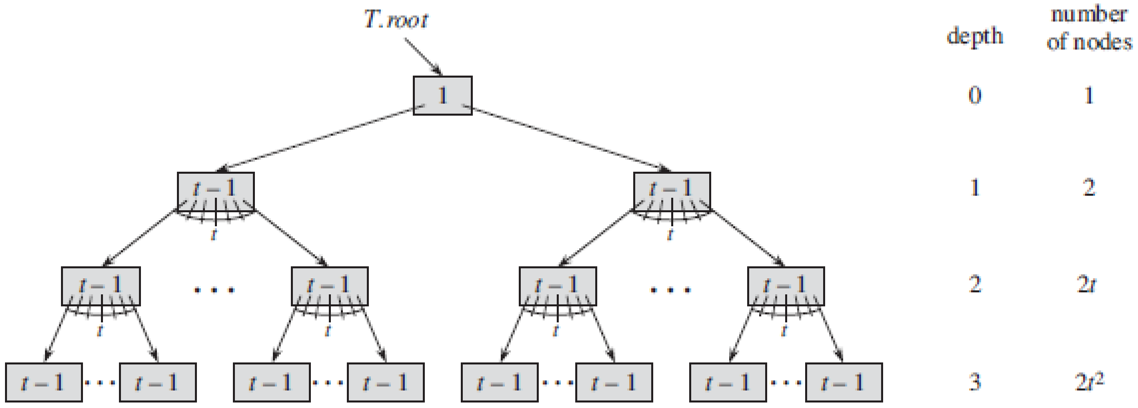
- Height
  - The number of disk accesses required for most operations on a B-tree is **proportional** to the **height** of the B-tree
  - Theorem

If $n \geq 1$, then for any n-key B-tree T of height h and minimum $\boxed{\text{degree}}$ $t \geq 2$.

$$h \leq \log_t \frac{n+1}{2} .$$

B-tree – Height

- **Height**
  - Proof
    - The root of a B-tree T contains at least 1 key
    - All the other nodes contain **at least** (t-1) keys
    - → T (with height = h) has
      - At least 2 nodes at depth 1
      - At least 2t nodes at depth 2
      - At least $2t^2$ nodes at depth 3, …, until at depth h, it has at least $2t^{h-1}$ nodes.
    - → number n of **keys** satisfies the inequality:



$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\ &= 2t^h - 1 . \end{aligned}$$

Superior or equal because it is **at least** t-1 keys per node!

$\longrightarrow$ $t^h \leq (n+1)/2.$ $\longrightarrow$ $h \leq \log_t \frac{n+1}{2}$ $\square$

B-tree – Main functions

- Search
- Insert, Delete …
  - More complicated functions with multiple cases
  - Insert
    - Check a node doesn't get too big
  - Remove
    - Check a node doesn't get too small

- Implementation
  - Remark
    - Previous Trees: simple definition of the node
    - Btree: 2 classes !
      - Btree (main data structure)
      - BTreeNode

B-tree – Main functions

- Pseudo-code
  - Search
    - With linear search

```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
    i = i + 1
if i ≤ x.n and k == x.key_i
    return (x, i)
elseif x.leaf
    return NIL
else DISK-READ(x.c_i)
    return B-TREE-SEARCH(x.c_i, k)
```

B-tree – Main functions

- Pseudo-code
  - Create an empty B-tree

```
B-TREE-CREATE(T)
    x = ALLOCATE-NODE()
    x.leaf = TRUE
    x.n = 0
    DISK-WRITE(x)
    T.root = x
```

B-tree – Main functions - Insert

- Insert the new key into an existing leaf node.
- We cannot insert a key into a leaf node that is full !
- → operation ***splits*** a full node y (having 2t-1 keys) around its ***median key*** y.key$_t$ into 2 nodes having only (t-1) keys each.
- The median key moves **up** into y's parent to identify the dividing point between the 2 new trees.
- But if y's parent is also full → must split it **before** we can insert the new key
- → we could end up splitting full nodes all the way up the tree !!
- Like a BST, we can insert a key into a B-tree in a single pass **down** the tree from the root to a leaf.
- Not wait to find out whether we will actually need to split a full node in order to do the insertion.
- Instead, travel **down** the tree searching for the position where the new key belongs
- Split each full node we come to along the way (including the leaf itself)
- → whenever we want to split a full node y, we are assured that its parent is not full.

B-tree – Main functions - Insert

- Main pseudo-code
  - O(h) disk accesses

```
B-TREE-INSERT(T, k)
r = T.root
if r.n == 2t − 1
    s = ALLOCATE-NODE()
    T.root = s
    s.leaf = FALSE
    s.n = 0
    s.c_1 = r
    B-TREE-SPLIT-CHILD(s, 1)
    B-TREE-INSERT-NONFULL(s, k)
else B-TREE-INSERT-NONFULL(r, k)
```

B-tree – Main functions - Insert

- Insert
  1. Initialize x as root.
  2. While (x is not leaf) do
     - Find the child of x that is going to be traversed next. Let the child be y.
     - If (y is not full) change x to point to y.
     - If (y is full) split it and change x to point to one of the 2 parts of y.
       - If (k < mid key) in y then set x as 1st part of y
       - Else second part of y.
       - When we **split** y, we move a key from y to its parent x.
  3. The loop in step 2 stops when x is leaf.
     - x must have space for 1 extra key! because we have been splitting all nodes in advance.
     - → insert k to x.

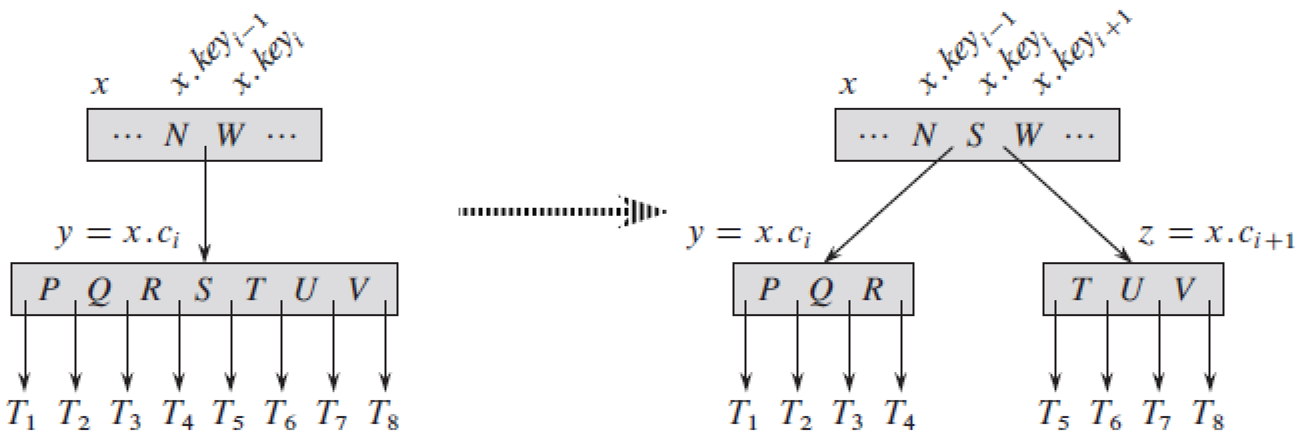
B-tree – Main functions - Split

- Input
  - a non full internal node x (already allocated in memory)
  - an index i | $x.c_i$ is a full child of x.
- Procedure
  - **splits** this child in 2 and adjusts x so that it has an additional child.
  - To split a full root, we will first make the root a child of a new empty root node, so that we can use B-TREE-SPLIT-CHILD.
- → The tree grows in height by one
  - **splitting is the only means by which the tree grows**.

B-tree – Main functions - Split

- Split

```
B-TREE-SPLIT-CHILD(x, i)
z = ALLOCATE-NODE()
y = x.c_i
z.leaf = y.leaf
z.n = t - 1
for j = 1 to t - 1
    z.key_j = y.key_{j+t}
if not y.leaf
    for j = 1 to t
        z.c_j = y.c_{j+t}
y.n = t - 1
for j = x.n + 1 downto i + 1
    x.c_{j+1} = x.c_j
x.c_{i+1} = z
for j = x.n downto i
    x.key_{j+1} = x.key_j
x.key_i = y.key_t
x.n = x.n + 1
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```

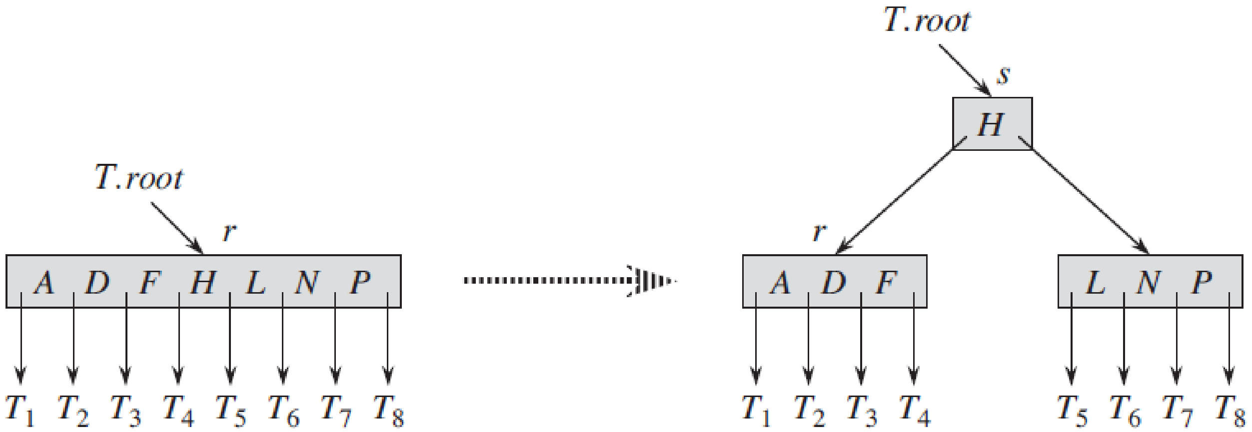


Splitting a node with t=4. Node y=x.c_i splits into two nodes, y and z, and the median key S of y moves up into y's parent

2t-1=7

B-tree – Main functions - Split

- Split
  - Example:



Splitting the root with t=4.

- Root node r splits in 2
- A new root node s is created.
  - The new root:
    - a) contains the median key of r
    - b) has the 2 halves of r as children
- The B-tree height +=1 when the root is split.

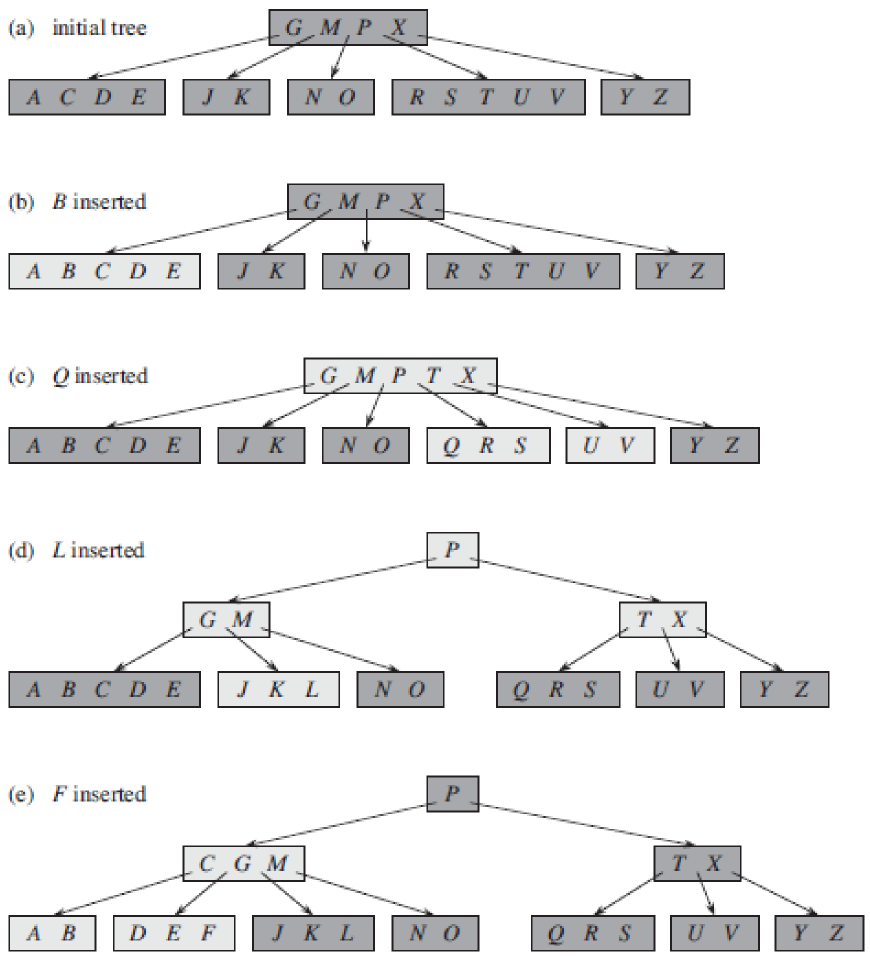
B-tree – Main functions – Insert nonfull

- Pseudo-code

```
B-TREE-INSERT-NONFULL(x, k)
i = x.n
if x.leaf
    while i ≥ 1 and k < x.key_i
        x.key_{i+1} = x.key_i
        i = i − 1
    x.key_{i+1} = k
    x.n = x.n + 1
    DISK-WRITE(x)
else while i ≥ 1 and k < x.key_i
        i = i − 1
    i = i + 1
    DISK-READ(x.c_i)
    if x.c_i.n == 2t − 1
        B-TREE-SPLIT-CHILD(x, i)
        if k > x.key_i
            i = i + 1
    B-TREE-INSERT-NONFULL(x.c_i, k)
```

B-tree – Main functions - Insert

- Example:



B-tree – Main functions – Delete

- Delete the key k
  - Constraints
    - Must guard against deletion producing a tree whose structure violates the B-tree properties
      - Check a node doesn't get too small during deletion!

B-tree – Main functions – Delete

- Delete the key k (main function)
  1. If k is in node x and x is a leaf → delete the key k from x.
  2. If it is in node x and x is an internal node, →do :
     a) If the child y that precedes k in node x has at least t keys, then find the predecessor k0 of k in the sub-tree rooted at y. Recursively delete k0, and replace k by k0 in x.
        can find k0 and delete it in a single downward pass.
     b) If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k0 of k in the subtree rooted at z. Recursively delete k0, and replace k by k0 in x. (We can find k0 and delete it in a single downward pass.)
     c) Else, if both y and z have only t-1 keys, merge k and all of z into y, so x loses both k and the pointer to z,
        y has now contains 2t-1 keys. Then free z and recursively delete k from y…
  3. If k is not present in internal node x, determine the root x.c(i) of the appropriate subtree that must contain k (if k is in the tree at all)
     If x.c(i) has only (t-1) keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys.
     - Then finish by recursing on the appropriate child of x
     a) If x.c(i) has only t-1 keys but has an immediate sibling with at least t keys, give x.c(i) an extra key by moving a key from x down into x.c(i), moving a key from x.c(i) 's immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into x.c(i).
     b) if x.c(i) and both of x.c(i)'s immediate siblings have t-1 keys, merge x.c(i) with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

B-tree – Main functions – Delete

- Cases
  1. **If** the key k is in node x and x is a leaf **then** delete the key k from x.
  2. **If** the key k is in node x and x is an internal node **then**
     a) **If** the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y.

        Recursively delete k', and replace k by k' in x.

        (find k' and delete it in a single downward pass)

     b) **If** y has fewer than t keys, **then**, symmetrically, examine the child z that follows k in node x.

        If z has at least t keys, **then** find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x.

        (can find k' and delete it in a single downward pass)

     c) **Else, if** both y and z have only (t -1) keys, merge k and all of z into y,

        so that x loses both k and the pointer to z, and y contains 2t -1 keys.

        Then free z, and recursively delete k from y.

B-tree – Main functions – Delete

- Cases

  3. **If** the key k is not present in internal node x, determine the root $x.c_i$ of the appropriate subtree that must contain k if k is in the tree at all.

     If $x.c_i$ has only (t-1) keys, execute step **3a** or **3b** as necessary to guarantee that we descend to a node containing at least t keys.

     Then finish by recursing on the appropriate child of x.

     a) If $x.c_i$ has only (t-1) keys but has an immediate sibling with at least t keys

        give $x.c_i$ an extra key by moving a key from x down into $x.c_i$,

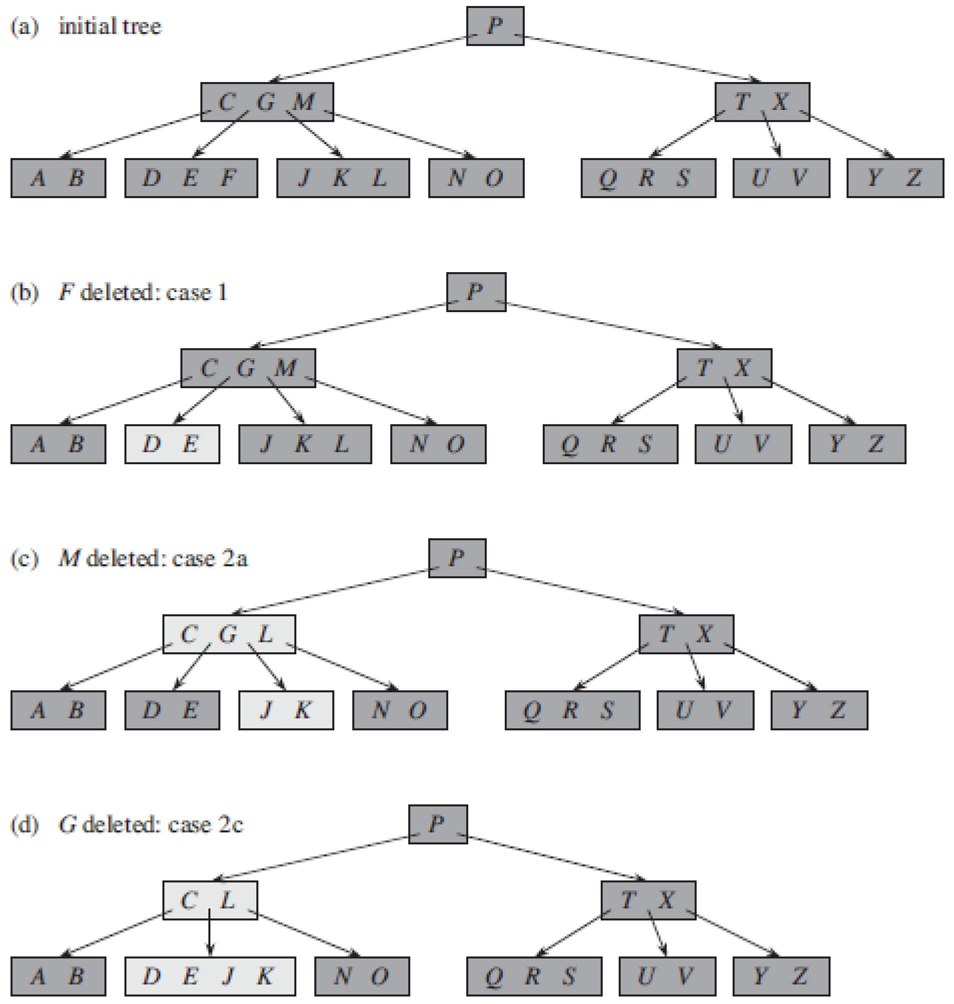
        moving a key from $x:c_i$'s immediate left or right sibling up into x

        and moving the appropriate child pointer from the sibling into $x.c_i$.

     b) If x.cand both of $x.c_i$'s immediate siblings have (t-1) keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the **median** key for that node.

B-tree – Main functions – Delete

- Example:
  - ➢ The minimum degree for this B-tree is t=3
  - ➢ → a node (diff than the root) cannot < than 2 keys
    - (modified nodes = lightly shaded)
  - ➢ Deletion of F:
    - **case 1**: simple deletion from a leaf
  - ➢ Deletion of M:
    - **case 2a**: the predecessor L of M moves **up** to take M's position.
  - ➢ Deletion of G:
    - **case 2c**: push G **down** to make node "DEGJK" and then delete G from this leaf.



B-tree – C++ implementation

- B-tree Class
  - Only 3 main methods !

```
class BTree {
    BTreeNode *root; // Pointer to root node
    int t;           // Minimum degree
public:
    // Constructor (Initializes tree as empty)
    BTree(int _t) { root = NULL; t = _t; }
    void traverse() { if (root != NULL) root->traverse(); }
    // search a key in this tree
    BTreeNode* search(int k) { return (root == NULL) ? NULL : root->search(k); }
    // insert a new key in the B-Tree
    void insert(int k);
    // remove a key in the B-Tree
    void remove(int k);
};
```

B-tree – C++ implementation

- BTreeNode Class
  - Main part of the data structure

```cpp
class BTreeNode {
    int *keys;      // Array of keys
    int t;          // Minimum degree (defines the range for number of keys)
    BTreeNode **C;  // An array of child pointers
    int n;          // Current number of keys
    bool leaf;      // true == node is leaf
public:
    BTreeNode(int _t, bool _leaf);   // Constructor
    // traverse all nodes in a subtree rooted with this node
    void traverse();
    // search a key in subtree rooted with this node, returns NULL if k is not present
    BTreeNode *search(int k);
    // returns the index of the first key >= k
    int findKey(int k);
    // insert a new key in the subtree rooted with this node
    // the node must be non-full when insertNonFull is called
    void insertNonFull(int k);
    // split the child y of this node. i= index of y in child array C[].  child y must be full when splitchild is called
    void splitChild(int i, BTreeNode *y);
    // remove the key k in subtree rooted with this node
    void remove(int k);
    // remove the key present in idx-th position in this node which is a leaf
    void removeFromLeaf(int idx);
    // remove the key present in idx-th position in this node which is a non-leaf node
    void removeFromNonLeaf(int idx);
    // get the predecessor of the key present in the idx-th position in the node
    int getPred(int idx);
    // get the successor of the key present in the idx-th position in the node
    int getSucc(int idx);
    // fill up the child node present in the idx-th position in the C[] array if that child has less than t-1 keys
    void fill(int idx);
    // borrow a key from the C[idx-1]-th node and place it in C[idx]th node
    void borrowFromPrev(int idx);
    // borrow a key from the C[idx+1]-th node and place it in C[idx]th node
    void borrowFromNext(int idx);
    // merge idx-th child of the node with idx+1 th child of the node
    void merge(int idx);
    // Set BTree friend of this --> access private members of BTreeNode in BTree methods
    friend class BTree;
};
```

B-tree – C++ implementation

- B tree methods
  - Insert function

```
// The main function that inserts a new key in this B-Tree
void BTree::insert(int k) {
    if (root == NULL) {
        root = new BTreeNode(t, true);
        root->keys[0] = k;  // Insert key
        root->n = 1;  // Update number of keys in root
    }
    else {
        // If root is full, then tree grows in height
        if (root->n == 2 * t - 1) {
            BTreeNode *s = new BTreeNode(t, false);
            s->C[0] = root; // set old root as child of new root
                            // split the old root and move 1 key to the new root
            s->splitChild(0, root);
            // new root has 2 children.
            //Decide which of the 2 children is going to have new key!
            int i = 0;
            if (s->keys[0] < k)
                i++;
            s->C[i]->insertNonFull(k);
            // Change root
            root = s;
        }
        else  // root is not full, call insertNonFull for root
            root->insertNonFull(k);
    }
}
```

B-tree – C++ implementation

- B tree methods
  - Remove function

```
void BTree::remove(int k) {
    if (!root) {
        cout << "The tree is empty.\n";
        return;
    }
    // Call the remove function for root
    root->remove(k);
    // if the root node has 0 keys --> its 1st child as the new root if it has a child
    // otherwise set root as NULL
    if (root->n == 0) {
        BTreeNode *tmp = root;
        if (root->leaf)
            root = NULL;
        else
            root = root->C[0];
        delete tmp; // free old root
    }
    return;
}
```

B-tree – C++ implementation

- BTreeNode methods
  - Creation of a node
  - Find key
  - Traverse
  - Search

```cpp
BTreeNode::BTreeNode(int t1, bool leaf1) {
    // copy the given minimum degree and leaf property !
    t = t1;  leaf = leaf1;
    // Allocate memory for maximum number of possible keys and child pointers
    keys = new int[2 * t - 1];
    C = new BTreeNode *[2 * t];
    n = 0; // Set  the number of keys to 0
}

// returns the index of the 1st key >= k
int BTreeNode::findKey(int k) {
    int idx = 0;
    while (idx<n && keys[idx] < k)
        ++idx;
    return idx;
}
```

```cpp
// traverse all the nodes in a subtree rooted with this node
void BTreeNode::traverse() {
    // There are n keys and n+1 children, traverse through n keys and 1st n children
    int i;
    for (i = 0; i < n; i++) {
        // if not leaf then before printing key[i]
        // traverse the subtree rooted with child C[i]
        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }
    if (leaf == false)
        C[i]->traverse();
}

// search key k in subtree rooted with this node
BTreeNode *BTreeNode::search(int k) {
    int i = 0;
    while (i < n && k > keys[i])
        i++;
    if (keys[i] == k)
        return this;
    if (leaf == true)
        return NULL;
    return C[i]->search(k);
}
```

B-tree – C++ implementation

- Remove

```
// remove the key k from the sub-tree rooted with this node
void BTreeNode::remove(int k) {
    int idx = findKey(k);
    if (idx < n && keys[idx] == k) { // k present in this node
        if (leaf)
            removeFromLeaf(idx);
        else
            removeFromNonLeaf(idx);
    }
    else {
        if (leaf) {
            cout << "The key " << k << " is does not exist.\n";
            return;
        }
        // The key to be removed is in the sub-tree rooted with this node
        // flag = whether the key is present in the sub-tree rooted with the last child of this node
        bool flag = ((idx == n) ? true : false);
        // If the child where the key is supposed to exist has less than t keys, fill that child
        if (C[idx]->n < t)
            fill(idx);
        // If the last child has been merged, must have merged with the previous child
        // we keep going on the idx-1 th child., otherwise we go on the idx th child (now has at least t keys)
        if (flag && idx > n)
            C[idx - 1]->remove(k);
        else
            C[idx]->remove(k);
    }
    return;
}
```

```
void BTreeNode::removeFromLeaf(int idx) {
    // Move all the keys after the idx-th pos one place backward
    for (int i = idx + 1; i<n; ++i)
        keys[i - 1] = keys[i];
    // Reduce the count of keys
    n--;
    return;
}

// A function to remove the idx-th key from this node - which is a non-leaf node
void BTreeNode::removeFromNonLeaf(int idx) {
    int k = keys[idx];
    // If the child that precedes k (C[idx]) has atleast t keys,
    // find the predecessor 'pred' of k in the subtree rooted at
    // C[idx]. Replace k by pred. Recursively delete pred
    // in C[idx]
    if (C[idx]->n >= t) {
        int pred = getPred(idx);
        keys[idx] = pred;
        C[idx]->remove(pred);
    }
    // If the child C[idx] has less that t keys, examine C[idx+1].
    // If C[idx+1] has atleast t keys, find the successor 'succ' of k in
    // the subtree rooted at C[idx+1]
    // Replace k by succ
    // Recursively delete succ in C[idx+1]
    else if (C[idx + 1]->n >= t) {
        int succ = getSucc(idx);
        keys[idx] = succ;
        C[idx + 1]->remove(succ);
    }
    // If both C[idx] and C[idx+1] has less that t keys,merge k and all of C[idx+1]
    // into C[idx]
    // Now C[idx] contains 2t-1 keys
    // Free C[idx+1] and recursively delete k from C[idx]
    else {
        merge(idx);
        C[idx]->remove(k);
    }
    return;
}
```

B-tree – C++ implementation

- Remove
  - Utility functions

```
// merge C[idx] with C[idx+1] C[idx+1] is freed after merging
void BTreeNode::merge(int idx) {
    BTreeNode *child = C[idx];
    BTreeNode *sibling = C[idx + 1];
    // Pulling a key from the current node and inserting it into (t-1)th position of C[idx]
    child->keys[t - 1] = keys[idx];
    // Copying the keys from C[idx+1] to C[idx] at the end
    for (int i = 0; i<sibling->n; ++i)
        child->keys[i + t] = sibling->keys[i];
    // Copying the child pointers from C[idx+1] to C[idx]
    if (!child->leaf) {
        for (int i = 0; i <= sibling->n; ++i)
            child->C[i + t] = sibling->C[i];
    }
    // Moving all keys after idx in the current node one step before -
    // to fill the gap created by moving keys[idx] to C[idx]
    for (int i = idx + 1; i<n; ++i)
        keys[i - 1] = keys[i];
    // Moving the child pointers after (idx+1) in the current node one step before
    for (int i = idx + 2; i <= n; ++i)
        C[i - 1] = C[i];
    // Updating the key count of child and the current node
    child->n += sibling->n + 1;
    n--;
    delete(sibling); // free memory occupied by sibling
    return;
}
```

```
// A function to get predecessor of keys[idx]
int BTreeNode::getPred(int idx) {
    // Keep moving to the right most node until we reach a leaf
    BTreeNode *cur = C[idx];
    while (!cur->leaf)
        cur = cur->C[cur->n];
    return cur->keys[cur->n - 1]; // the last key of the leaf
}

int BTreeNode::getSucc(int idx) {
    // Keep moving the left most node starting from C[idx+1] until we reach a leaf
    BTreeNode *cur = C[idx + 1];
    while (!cur->leaf)
        cur = cur->C[0];
    return cur->keys[0]; //  the 1st key of the leaf
}

// fill child C[idx] which has less than t-1 keys
void BTreeNode::fill(int idx) {
    // If the previous child(C[idx-1]) more than t-1 keys --> borrow a key
    // from that child
    if (idx != 0 && C[idx - 1]->n >= t)
        borrowFromPrev(idx);
    // If the next child(C[idx+1]) more than t-1 keys --> borrow a key
    // from that child
    else if (idx != n && C[idx + 1]->n >= t)
        borrowFromNext(idx);
    // Merge C[idx] with its sibling
    else {
        if (idx != n) // If C[idx] == last child
            merge(idx); // merge with with its previous sibling
        else
            merge(idx - 1); // merge with its next sibling
    }
    return;
}
```

B-tree – C++ implementation

- Remove
  - Utility functions: Borrow from **Previous** and from **Next**

```cpp
// borrow a key from C[idx-1] and insert it into C[idx]
void BTreeNode::borrowFromPrev(int idx) {
    BTreeNode *child = C[idx];
    BTreeNode *sibling = C[idx - 1];
    // The last key from C[idx-1] goes up to the parent and key[idx-1]
    // from parent is inserted as the 1st key in C[idx]
    // --> the loses sibling one key and child gains one key
    // Moving all key in C[idx] one step ahead
    for (int i = child->n - 1; i >= 0; --i)
        child->keys[i + 1] = child->keys[i];
    // If C[idx] is not a leaf, move all its child pointers one step ahead
    if (!child->leaf) {
        for (int i = child->n; i >= 0; --i)
            child->C[i + 1] = child->C[i];
    }
    // Setting child's 1st key == to keys[idx-1] from the current node
    child->keys[0] = keys[idx - 1];
    // Moving sibling's last child as C[idx]'s 1st child
    if (!(child->leaf))
        child->C[0] = sibling->C[sibling->n];
    // Moving the key from the sibling to the parent
    // This reduces the number of keys in the sibling
    keys[idx - 1] = sibling->keys[sibling->n - 1];
    child->n += 1;
    sibling->n -= 1;
    return;
}
```

```cpp
// borrow a key from the C[idx+1] and place it in C[idx]
void BTreeNode::borrowFromNext(int idx) {
    BTreeNode *child = C[idx];
    BTreeNode *sibling = C[idx + 1];
    // keys[idx] is inserted as the last key in C[idx]
    child->keys[(child->n)] = keys[idx];
    // Sibling's 1st child is inserted as the last child into C[idx]
    if (!(child->leaf))
        child->C[(child->n) + 1] = sibling->C[0];
    // 1st key from sibling is inserted into keys[idx]
    keys[idx] = sibling->keys[0];
    // Moving all keys in sibling one step behind
    for (int i = 1; i<sibling->n; ++i)
        sibling->keys[i - 1] = sibling->keys[i];
    // Moving the child pointers one step behind
    if (!sibling->leaf) {
        for (int i = 1; i <= sibling->n; ++i)
            sibling->C[i - 1] = sibling->C[i];
    }
    // Increasing and decreasing the key count of C[idx] and C[idx+1]
    child->n += 1;
    sibling->n -= 1;
    return;
}
```

B-tree – C++ implementation

- Insert

```cpp
// insert a new key in this node, the node must be non-full when insertNonFull is called
void BTreeNode::insertNonFull(int k) {
    // Initialize index as index of rightmost element
    int i = n - 1;
    // If this is a leaf node
    if (leaf == true) {
        // 1) Finds the location of new key to be inserted
        // 2) Moves all greater keys to 1 place ahead
        while (i >= 0 && keys[i] > k) {
            keys[i + 1] = keys[i];
            i--;
        }
        // Insert the new key at found location
        keys[i + 1] = k;
        n = n + 1;
    }
    else { // If this node is not leaf
           // Find the child which is going to have the new key
        while (i >= 0 && keys[i] > k)
            i--;
        // See if the found child is full
        if (C[i + 1]->n == 2 * t - 1) {
            // If the child is full, then split it
            splitChild(i + 1, C[i + 1]);
            // After split, the middle key of C[i] goes up and
            // C[i] is splitted into 2
            // See which of the 2 is going to have the new key
            if (keys[i + 1] < k)
                i++;
        }
        C[i + 1]->insertNonFull(k);
    }
}
```

15

```cpp
// split the child y of this node (y must be full when splitChild is called)
void BTreeNode::splitChild(int i, BTreeNode *y) {
    // Create a new node which is going to store (t-1) keys of y
    BTreeNode *z = new BTreeNode(y->t, y->leaf);
    z->n = t - 1;
    // Copy the last (t-1) keys of y to z
    for (int j = 0; j < t - 1; j++)
        z->keys[j] = y->keys[j + t];
    // Copy the last t children of y to z
    if (y->leaf == false) {
        for (int j = 0; j < t; j++)
            z->C[j] = y->C[j + t];
    }
    // Reduce the number of keys in y
    y->n = t - 1;
    // As this node is going to have a new child, create space of new child
    for (int j = n; j >= i + 1; j--)
        C[j + 1] = C[j];
    // Link the new child to this node
    C[i + 1] = z;
    // A key of y will move to this node. Find location of
    // new key and move all greater keys one space ahead
    for (int j = n - 1; j >= i; j--)
        keys[j + 1] = keys[j];
    // Copy the middle key of y to this node
    keys[i] = y->keys[t - 1];
    // Increment count of keys in this node
    n = n + 1;
}
```

Conclusion

- B-trees
  - Efficient data structure
    - Keep keys in sorted order for sequential traversing
    - Use a hierarchical index to minimize the number of disk reads
    - Use partially full blocks to speed insertions and deletions
    - Keep the index balanced with a recursive algorithm
- Complexity
  - 2-3 tree & B-tree

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Questions ?

- Acknowledgment + Reading
  - Chapter 18, B-trees, Introduction to Algorithms, 3rd Edition.