**B-Trees**

First 10 slides are to recap the BST essentials
Quick Recap of Simple BSTs

Notation: \( n \): number of nodes, \( e \): number of external nodes, \( i \): number of internal nodes, \( h \): height.

Properties:
- \( e = i + 1 \)
- \( n = 2e - 1 \)
- \( h \leq i \)
- \( h \leq (n - 1)/2 \)
- \( e \leq 2^h \)
- \( h \geq \log_2 e \)
- \( h \geq \log_2 (n + 1) - 1 \)

Traversals:
- Preorder: visit, left, right
- Inorder: left, visit, right
- Postorder: left, right, visit

The depth of a node \( v \) is the number of ancestors of \( v \), excluding \( v \) itself. Note that this definition implies that the depth of the root of \( T \) is 0. The depth of a node \( v \) can also be recursively defined as follows: (1) If \( v \) is the root, then the depth of \( v \) is 0; (2) otherwise, the depth of \( v \) is one plus the depth of the parent of \( v \). The height of a node \( v \) is (1) 0, if \( v \) is an external node and (2) is one plus the maximum height of a child of \( v \).

Creation, Insertion, deletion, Look-up, Find_max, Find_min, and successor/predecessor (with respect to a specific traversal); they are all \( O(h) \) operations; in the worst case \( O(n) \).

Applications: Build trees for arithmetic expressions, print/evaluate expression trees, Range Queries, Trimming a BST

A full binary tree (sometimes proper binary tree or 2-tree or a strict tree) is a tree in which every node other than the leaves has two children.

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

They are all \( O(n) \) operations
A binary search tree is organized as a binary tree with some special property. We represent such a tree by a linked data structure in which each node is a `struct`. In addition to a `key` (for simplicity we assume it’s an integer; it can be anything just like in any kind of linked lists), each node contains two pointers `left` and `right`; usually no information is explicitly stored about the parent of a node; if left (right) child does not exist, the corresponding link contains a `NULL`.

```c
typedef struct node { struct node *left; int key; struct node *right; } node;
```

The keys in a binary search tree are always stored in such a way as to satisfy the binary-search-tree property: “Let `x` be a node in a binary search tree. If `y` is a node in the left subtree of `x`, then `y.key <= x.key`. If `y` is a node in the right subtree of `x`, then `y.key >= x.key`.”

We can have many different BSTs with the same set of nodes [computing how many is outside our scope in this class]. Any given BST can be specified by the address (pointer) of the root; an empty tree is represented by the `NULL` pointer (Create an empty BST using `t_node *Root = NULL;`)

**Observe:**
- Each node has a unique path to the root of the tree.
- The maximum of the length of all such paths is called the height of the tree.
- `# nodes = # edges + 1` (it is true for any tree, binary or not); **why?**
- In the worst case, height of the tree can be `n – 1`; example??

*Refreshing from 2120*
**Operations on BSTs**

**Set / Dictionary:** maintain dynamic collection of elements
- Insert (k) : insert key k
- Remove (k) : remove (delete) key k
- Find (k) : is key k present?
- Traversals [will soon define]

**Map:** maintain collection of (key, value) pairs
- Insert (k, v) : insert key k with associated value v
- Remove (k) : remove record with key k
- Find (k) : return value associated with key k (or report that k isn’t in the structure)

**How to store a BST?**
- We can maintain a BST by using a sorted/unsorted array, or linked (singly/doubly) list, a hash table [Later] and others – all depends on what we want to do in the applications.
- Usually, we use a struct like this: `typedef Struct node{struct node *left; int key; struct node *right;} node;` Remember, we can have many other info in place of key alone; say in stead of value [see above] we can have a pointer pointing to a record of a file.
  - Advantage: we can change the file structure without disturbing the BST.
- To create an empty tree, we use `node *root = NULL` [Just like a linked list, the pointer variable root is a gateway to the entire tree.]
- Sometimes, we can add another component to the structure like `struct node *parent` to keep the information of the parent of a node in a BST.

*Refreshing from 2120*
Search, Insert, Traversals*

**Search** a BST T (Tree root pointer) for a given key (Find the tree node containing key k) [will return the pointer to the desired node]: Execution time is O(n) in the worst case

- If the tree is empty or the key is not found, the function returns NULL.
- If the tree is not empty, then compare the search key k with the key of the node; if equal, search is successful; otherwise go down either the left link or right link depending on if k is less or ≥. One can easily write a recursive routine [or iterative routine in C or C++]

**Insert** a node with key k in T: Search for k; if found exit with an error; otherwise when search returns a NULL, create (malloc) a new node, adjust the fields. One can easily write a recursive routine [or iterative routine in C or C++]

**Tree Traversals**: Enumeration [visiting (printing) once] of the contents of the nodes of a BST in some order. (1) **Inorder**: left, root, right; (2) **Preorder**: root, left, right; (3) **Postorder**: left, right, root;

Pseudocode (recursive) for Inorder traversal:

```
Inorder(T):
if T == NULL, then return
Inorder(T->left)
print T->key
Inorder(T->right)
```

Let us see examples.

(A) 2 5 5 6 7 8  
(B) 2 5 5 6 7 8

*Refreshing from 2120*
Search, Insert, Max, Min, Traversals (in C)*

```c
node *search (node **lp, int key){
    node *current = *lp;
    while (current != NULL && current->info != key)
        if (key < current->info) current = current->left;
        else current = current->right;
    return (current); }// One can check if the key exists.

Note: One can maintain a stack of pointers to keep track of the nodes while searching. Will be useful later.

void insert (node **lp, int key){
    node *current, *prev, *new;
    current = *lp; prev = NULL;
    while (current != NULL) {
        prev = current;
        if (key < current->info) current = current->left;
        else current = current->right;
    }
    new = (node *)malloc(sizeof(node));
    new->info = key;
    new->left = new->right =NULL;
    if (prev == NULL) *lp = new; else {
        if (key < prev->info) prev->left = new;
        else prev->right = new;
    }
}

int min (node **lp){
    node *x =*lp;
    while (x->left != NULL)
        x = x->left;
    return (x->info);
} //Assuming the node is at least one node.

Note: int max is obtained by replacing left with right.

void inorder_recursive (node **lp){
    node *x = *lp;
    if (x != NULL)
        {inorder_recursive (&x->left);
        printf("%d ", x->info);
        inorder_recursive (&x->right);
        }
}

Note: preorder and post order traversals are similar.

Q: How to write non recursive traversals? [Use stacks to remember the path!]

*Refreshing from 2120
**Predecessor & Successor**

Predecessor (Successor) of any node in a given binary search tree is usually defined in terms of the unique inorder traversal of the tree; the node may be specified as a key or the address of the node (depending on application requirement) – accordingly, the function (method) has to be adjusted – the principle remains the same.

We’ll study some scenarios first to understand the logic.

- inorder traversal: -1 2 4 6 8 11 12 15 20 21 23 25
- Say, key is 4 ⇒ its right link is not null ⇒ 4’s successor must be the minimum of the node 8, which is 6.
- Say key is 11 ⇒ right link is null ⇒ its successor must be somewhere up in the tree, go to the predecessor, say node p ⇒ two cases:
  - Case 1: key is NOT on its predecessor’s right link ⇒ the key of p is key’s successor (if p is not null) [i.e., 12 is successor of 11]
  - Case 2: if p is not NULL and key is on its predecessor’s right link, go up the chain and repeat.
- Be careful to handle the maximum key in the BST, that does not have a successor

Writing a function (method) for successor is relatively straightforward – remember to maintain a stack of pointers when searching for the tree [necessary to move up]; actually, stack is not needed; one can do it without a stack.

**Note:** Predecessor is similar to successor – left links will replace the right links.

*Refreshing from 2120*
Successor of a node*

```c
int successor (node **lp, int key) {
    node *root = lp, *succ = NULL, *keynode = search (&root, key);
    // if right subtree is not null
    if (keynode->right != NULL) return (min (&keynode->right));
    // printf ("HERE min is \%d\n", min (&keynode->right));
    // Start from root and search down the tree
    while (root != NULL) {
        if (keynode->info < root->info) {
            succ = root; root = root->left;
        } else {
            if (keynode->info > root->info) root = root->right;
            else break;
        }
    }
    if (succ == NULL) return -100;
    else return succ->info;
}
```

*Note:* if one needs to have a function that returns the address of the successor of the info of the successor node, one needs to redefine the function as

```c
node *successor_node ((node **lp, int key);
```

Observe the same code will work up until the last if statement; the last if statement needs be replaced by return succ !!

One can adjust the code in a relatively straightforward way to get a function for the inorder predecessor; try that.

*Refreshing from 2120*
Removing a node from a BST is a bit more tricky, since we do want to make sure that the BST remains a BST after the deletion. If the node has one child then the child is spliced to the parent of the node. If the node has two children then its successor has no left child; copy the successor into the node and delete the successor instead. TREE-DELETE \((T, z)\) removes the node pointed to by \(z\) from the tree \(T\). There are 3 possibilities:

- **The node to be deleted is a leaf node:**
  Easiest: Just delete it; that’s all [Be careful when deleting the root]

- **Node to be deleted has only one child (left or right):**
  Connect the only subtree of the node to the parent of the node to be deleted and delete.

- **Node to be deleted has two children:**
  Tricky: Find inorder successor of the node. Copy contents of the inorder successor to the node (to be deleted) and delete the inorder successor.

Note that in the third case inorder successor of the node, say \(z\), (to be deleted) is the minimum of the node \(z\)-right.

*Refreshing from 2120*
## C Code

```c
void delete (node **lp, int key){
    node *root = *lp;
    /*succ = NULL, *keynode = search (&root, key);
*/ base case: the tree is empty.
    if (root == NULL) return;
    // if key is less than root, key is on root's left subtree
    if (key < root->info) delete (&root->left, key);
    // if key is greater than root, key is on root's right subtree
    else if (key > root->info) delete (&root->right, key);
    // if key is equal to root, root is to be deleted.
    else
        { //node with one or no child
            if (root->left == NULL) { *lp = root->right; free (root); return; }
            else if (root->right == NULL) { *lp = root->left; free (root); return; }
        }
    // node with two children; get inorder successor
    node *temp = minnode (&root->right);
    //copy the inorder successor's content to this node
    root->info = temp->info; *lp = root;
    delete(&root->right, temp->info);
}
```

We have used a helper function (a variation of the min function we have seen before.

```c
node *minnode (node **lp){
    node *x = *lp;
    while (x->left != NULL) x = x->left;
    return x;
}
```

**Note:** It is not very difficult; try to understand the logic as explained by the example and the comments embedded in the code. And yes, it is a recursive routine; the iterative version would be a bit more complicated. If you understand the logic, you should be able to do the deletions by hand on small examples. Practice with several arbitrary BSTs.

*Refreshing from 2120*
What is a B-tree & Why?

**What is a B-tree:** B-trees are balanced search trees designed to work well on disks or other direct access secondary storage devices. B-trees are like red-black trees, we have seen earlier, but they are better at minimizing disk I/O operations. Many database systems use B-trees, or variants of B-trees, to store information.

B-trees differ from red-black trees in that B-tree nodes may have many children, from a few to thousands. That is, the “branching factor” of a B-tree can be quite large, although it usually depends on characteristics of the disk unit used. B-trees are like red-black trees in that every n-node B-tree has height $O(\log n)$.

The exact height of a B-tree can be considerably less than that of a red-black tree, because its branching factor, and hence the base of the logarithm that expresses its height, can be much larger.

The goal is to get fast access to the data, and with disk drives this means reading a very small number of records. Note that a large node size (with lots of keys in the node) also fits with the fact that with a disk drive one can usually read a fairly large amount of data at once (perhaps 1024 bytes).

**To remember:** B-tree nodes have many more than two children. A B-tree node may contain more than just a single element.
B-Tree Rules

Every B-tree depends on a positive constant integer called MINIMUM, which is used to determine how many elements are held in a single node.

Rule 1: The root can have as few as one element (or even no elements if it also has no children); every other node has at least MINIMUM elements.

Rule 2: The maximum number of elements in a node is twice the value of MINIMUM.

Rule 3: The elements of each B-tree node are stored in a partially filled array, sorted from the smallest element (at index 0) to the largest element (at the final used position of the array).

Rule 4: The number of subtrees below a non-leaf node is always one more than the number of elements in the node.

Rule 5: For any non-leaf node:
- An element at index i is greater than all the elements in subtree number i of the node, and
- An element at index i is less than all the elements in subtree number i + 1 of the node.

Rule 6: Every leaf in a B-tree has the same depth. Thus, it ensures that a B-tree avoids the problem of an unbalanced tree.

Remember: "Every child of a node is also the root of a smaller B-tree".

12
Small Examples

Elements in subtree 0 is less than 93

Elements in subtree 1 is between 93 and 107

Elements in subtree 2 is greater than 107

MINIMUM = 1
Observations

Each internal node of a B-tree will contain several keys. Usually, the number of keys is chosen to vary between $d$ and $2d + 1$. In practice, the keys take up the most space in a node. The factor of 2 will guarantee that nodes can be split or combined. If an internal node has keys, then adding a key to that node can be accomplished by splitting the key node into two key nodes and adding the key to the parent node.

Each split node has the required minimum number of keys. Similarly, if an internal node and its neighbor each have keys, then a key may be deleted from the internal node by combining with its neighbor.

Deleting the key would make the internal node have keys; joining the neighbor would add keys plus one more key brought down from the neighbor's parent. The result is an entirely full node of $2d$ keys.

A B-tree is kept balanced by requiring that all leaf nodes are at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more node further away from the root.

B-trees have substantial advantages over alternative implementations when node access times far exceed access times within nodes, because then the cost of accessing the node may be amortized over multiple operations within the node. This usually occurs when the nodes are in secondary storage such as disk drives. By maximizing the number of child nodes within each internal node, the height of the tree decreases, and the number of expensive node accesses is reduced. \textit{In addition, rebalancing the tree occurs less often.}
Variants of B-trees

The term B-tree may refer to a specific design or it may refer to a general class of designs. In the narrow sense, a B-tree stores keys in its internal nodes but need not store those keys in the records at the leaves. The general class includes variations such as the B+-tree and the B*-tree.

- In the B+-tree, copies of the keys are stored in the internal nodes; the keys and records are stored in leaves; in addition, a leaf node may include a pointer to the next leaf node to speed sequential access.

- The B*-tree balances more neighboring internal nodes to keep the internal nodes more densely packed. This variant requires non-root nodes to be at least 2/3 full instead of 1/2.

- To maintain this, instead of immediately splitting up a node when it gets full, its keys are shared with a node next to it. When both nodes are full, then the two nodes are split into three.

- **Counted B-trees** store, with each pointer within the tree, the number of nodes in the subtree below that pointer. This allows rapid searches for the Nth record in key order or counting the number of records between any two records, and various other related operations.

**Note:** Etymology unknown: The origin of the name "B-tree" has never been explained by the original authors.
Definition of B-trees

A B-tree $T$ is a rooted tree (whose root is $T.root$) having the following properties:

1. Every node $x$ has the following attributes:
   
a. $x.n$, the number of keys currently stored in node $x$,
b. the $x.n$ keys themselves, $x.key_1, x.key_2, \ldots, x.key_{x.n}$, stored in nondecreasing order, so that $x.key_1 \leq x.key_2 \leq \cdots \leq x.key_{x.n}$,
c. $x.leaf$, a boolean value that is TRUE if $x$ is a leaf and FALSE if $x$ is an internal node.

2. Each internal node $x$ also contains $x.n + 1$ pointers $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$ to its children. Leaf nodes have no children, and so their $c_i$ attributes are undefined.

3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if $k_i$ is any key stored in the subtree with root $x.c_i$, then
   
   $$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \cdots \leq x.key_{x.n} \leq k_{x.n+1}$$

4. All leaves have the same depth, which is the tree’s height $h$.

5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \geq 2$ called the minimum degree of the B-tree:
   
a. Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least $t$ children. If the tree is nonempty, the root must have at least one key.
b. Every node may contain at most $2t - 1$ keys. Therefore, an internal node may have at most $2t$ children. We say that a node is full if it contains exactly
**Definition of B-trees**

Terminology: Unfortunately, the literature on B-trees is not uniform in its use of terms relating to B-Trees. We will use Knuth’s definition. For simplicity, most authors assume there are a fixed number of keys that fit in a node. The basic assumption is the key size is fixed, and the node size is fixed. In practice, variable length keys may be employed.

According to Knuth's definition, a B-tree of order \( m \) (the maximum number of children for each node) is a tree which satisfies the following properties:

1. Every node has at most \( m \) children.
2. Every node (except root) has at least \( \frac{m}{2} \) children.
3. The root has at least two children if it is not a leaf node.
4. All leaves appear in the same level, and carry information.
5. A non-leaf node with \( k \) children contains \( k-1 \) keys.

\( m \) is the upper bound on the number of
Definition of $B$-trees

A $B$-tree $T$ is a rooted tree (whose root is $T$::root) having the following properties:

1. Every node $x$ has the following attributes:
   a. $x$:$n$, the number of keys currently stored in node $x$,
   b. the $x$:$n$ keys themselves, $x$::key$_1$, $x$::key$_2$, ..., $x$::key$_{x:n}$, stored in nondecreasing order, so that $x$::key$_1 \leq x$::key$_2 \leq \cdots \leq x$::key$_{x:n}$,
   c. $x$::leaf, a boolean value that is TRUE if $x$ is a leaf and FALSE if $x$ is an internal node.

2. Each internal node $x$ also contains $x$:$n + 1$ pointers $x$::$c_1$, $x$::$c_2$, ..., $x$::$c_{x:n+1}$ to its children. Leaf nodes have no children, and so their $c_i$ attributes are undefined.
Definition of B-trees (contd.)

3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if $k_i$ is any key stored in the subtree with root $x.c_i$, then

$$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \cdots \leq x.key_{x.n} \leq k_{x.n+1}.$$

4. All leaves have the same depth, which is the tree’s height $h$.

5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \geq 2$ called the minimum degree of the B-tree:
   a. Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least $t$ children. If the tree is nonempty, the root must have at least one key.
   b. Every node may contain at most $2t - 1$ keys. Therefore, an internal node may have at most $2t$ children. We say that a node is full if it contains exactly $2t - 1$ keys.\(^2\)

The simplest B-tree occurs when $t = 2$. Every internal node then has either 2, 3, or 4 children, and we have a 2-3-4 tree. In practice, however, much larger values of $t$ yield B-trees with smaller height.
The Height of a B-tree

The number of disk accesses required for most operations on a B-tree is proportional to the height of the B-tree. We now analyze the worst-case height of a B-tree.

Theorem: If \( n \geq 1 \), then for any \( n \)-key B-tree \( T \) of height \( h \) and minimum degree \( t \geq 2 \),

\[
h \leq \log_t \frac{n+1}{2}
\]

Proof: The root of a B-tree \( T \) contains at least one key, and all other nodes contain at least \( t - 1 \) keys. Thus \( T \), whose height is \( h \), has at least 2 nodes at depth 1, at least \( 2t \) nodes at depth 2, at least \( 2t^2 \) nodes at depth 3, and so on, until at depth \( h \) it has at least \( 2^{t^{h-1}} \) nodes. Thus, the number \( n \) of keys satisfies the inequality

\[
n \geq 1 + (t - 1) \sum_{i=1}^{h} 2t^{i-1}
\]

\[
= 1 + 2(t - 1) \left( \frac{t^h - 1}{t - 1} \right)
\]

\[
= 2t^h - 1.
\]

By simple algebra, we get \( t^h \leq (n + 1)/2 \). Taking base-\( t \) logarithms of both sides proves the theorem. \( \blacksquare \)
number $n$ of keys satisfies the inequality
\[ n \geq 1 + (t - 1) \sum_{i=1}^{h} 2t^{i-1} \]
\[ = 1 + 2(t - 1) \left( \frac{t^h - 1}{t - 1} \right) \]
\[ = 2t^h - 1. \]

By simple algebra, we get $t^h \leq (n + 1)/2$. Taking base-$t$ logarithms of both sides proves the theorem.

Here we see the power of B-trees, as compared with red-black trees. Although the height of the tree grows as $O(\lg n)$ in both cases (recall that $t$ is a constant), for B-trees the base of the logarithm can be many times larger. Thus, B-trees save a factor of about $\lg t$ over red-black trees in the number of nodes examined for most tree operations. Because we usually must access the disk to examine an arbitrary node in a tree, B-trees avoid a substantial number of disk accesses.
Example

A B-tree of height 3 containing a minimum possible number of keys. Shown inside each node $x$ is $x.n$. 
Basic operations on B-trees

In this section, we present the details of the operations B-TREE-SEARCH, BTREE-CREATE, and B-TREE-INSERT. In these procedures, we adopt two conventions:

- The root of the B-tree is always in main memory, so that we never need to perform a DISK-READ on the root.
- We do have to perform a DISK-WRITE of the root, however, whenever the root node is changed.
- Any nodes that are passed as parameters must already have had a DISK-READ operation performed on them.

Our procedures are all “one-pass” algorithms that proceed downward from the root of the tree, without having to back up.
Searching a B-tree

Searching a B-tree is much like searching a binary search tree – binary replaced by multiway branching.

B-TREE-SEARCH takes as input a pointer to the root node x of a subtree and a key k to be searched for in that subtree. The top-level call is thus of the form B-TREE-SEARCH(T.root, k). If k is in the B-tree,

B-TREE-SEARCH returns the ordered pair (y, i) consisting of a node y and an index i such that y.key_i = k. Otherwise, the procedure returns NIL.

B-TREE-SEARCH(x, k)
1  i = 1
2  while i ≤ x.n and k > x.key_i
3     i = i + 1
4  if i ≤ x.n and k == x.key_i
5     return (x, i)
6  elseif x.leaf
7     return NIL
8  else DISK-READ(x.c_i)
9     return B-TREE-SEARCH(x.c_i, k)
Observation

As in the TREE-SEARCH procedure for binary search trees, the nodes encountered during the recursion form a simple path downward from the root of the tree. The B-TREE-SEARCH procedure therefore accesses \( O(h) = O(\log_t n) \) disk pages, where \( h \) is the height of the B-tree and \( n \) is the number of keys in the B-tree.

Since \( x.n < 2t \), the while loop of lines 2–3 takes \( O(t) \) time within each node, and the total CPU time is \( O(th) D = O(t \log_t n) \).
Creating an Empty B-tree

To build a B-tree $T$, we first use B-TREE-CREATE to create an empty root node and then call B-TREE-INSERT to add new keys. Both procedures use an auxiliary procedure ALLOCATE-NODE, which allocates one disk page to be used as a new node in $O(1)$ time. We can assume that a node created by ALLOCATE-NODE requires no DISK-READ, since there is yet no useful information stored on the disk for that node.

```
B-TREE-CREATE($T$)
1  $x = \text{ALLOCATE-NODE}()$
2  $x.leaf = \text{true}$
3  $x.n = 0$
4  DISK-WRITE($x$)
5  $T.root = x$
```

B-TREE-CREATE requires $O(1)$ disk operations and $O(1)$ CPU time.
Inserting a key into a B-tree

Inserting a key into a B-tree is significantly more complicated than inserting a key into a binary search tree.

We search for the leaf position at which to insert the new key. With a B-tree, however, we cannot simply create a new leaf node and insert it, as the resulting tree would fail to be a valid B-tree.

Instead, we insert the new key into an existing leaf node. Since we cannot insert a key into a full leaf node, we need an operation that splits a full node \( y \) (having \( 2t - 1 \) keys) around its median key \( y.key_t \) into two nodes having only \( t - 1 \) keys each. The median key moves up into \( y \)’s parent to identify the dividing point between the two new trees.

But if \( y \)’s parent is also full, we must split it before we can insert the new key, and thus we could end up splitting full nodes all the way up the tree.

We insert a key into a B-tree in a single pass down the tree from the root to a leaf. As we travel down the tree searching for the position where the new key belongs, we split each full node we come to along the way (including the leaf itself). Thus, whenever we want to split a full node \( y \), we are assured that its parent is not full.

We need a **splitting** procedure.
Splitting a node in a B-tree

The procedure B-TREE-SPLIT-CHILD takes as input a non-full internal node x (assumed to be in main memory) and an index i such that x.c_i (also assumed to be in main memory) is a full child of x. The procedure then splits this child in two 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 thus grows in height by one; splitting is the only means by which the tree grows.

See the figure below. We split the full node y = x.c_i about its median key S, which moves up into y’s parent node x. Those keys in y that are greater than the median key move into a new node z, which becomes a new child of x.

Splitting a node with t = 4. Node y D x.c_i splits into two nodes, y and z, and the median key S of y moves up into y’s parent.
The CPU time used by B-TREE-SPLIT-CHILD is $\Theta(t)$, due to the loops on lines 5–6 and 8–9. (The other loops run for $O(t)$ iterations.) The procedure performs $O(t)$ disk operations.
Inserting a key into a B-tree in a single pass

We insert a key \( k \) into a B-tree \( T \) of height \( h \) in a single pass down the tree, requiring \( O(h) \) disk accesses. The CPU time required is \( O(th) = O(t \log_t n) \). The B-TREE-INSERT procedure uses B-TREE-SPLIT-CHILD to guarantee that the recursion never descends to a full node.

\[
\text{B-TREE-INSERT}(T, k)
\]

1. \( r = T.\text{root} \)
2. \( \text{if } r.n == 2t - 1 \)
3. \( s = \text{ALLOCATE-NODE}() \)
4. \( T.\text{root} = s \)
5. \( s.\text{leaf} = \text{FALSE} \)
6. \( s.n = 0 \)
7. \( s.c_1 = r \)
8. \( \text{B-TREE-SPLIT-CHILD}(s, 1) \)
9. \( \text{B-TREE-INSERT-NONFULL}(s, k) \)
10. \( \text{else } \text{B-TREE-INSERT-NONFULL}(r, k) \)

Notes:

1. Lines 3–9 handle the case in which the root node \( r \) is full: the root splits and a new node \( s \) (having two children) becomes the root. Splitting the root is the only way to increase the height of a B-tree.

2. The auxiliary recursive procedure B-TREE-INSERT-NONFULL inserts key \( k \) into node \( x \), which is assumed to be non-full when the procedure is called. The operation of B-TREE-INSERT and the recursive operation of B-TREE-INSERT-NONFULL guarantee that this assumption is true.
**B-TREE-SPLIT-CHILD** \((x, i)\)

**B-TREE-SPLIT-CHILD** \((x, i)\)

1. \(z = \text{ALLOCATE-NODE}()\)
2. \(y = x.c_i\)
3. \(z.\text{leaf} = y.\text{leaf}\)
4. \(z.n = t - 1\)
5. for \(j = 1\) to \(t - 1\)
6. \(z.key_j = y.key_{j+1}\)
7. if not \(y.\text{leaf}\)
8. for \(j = 1\) to \(t\)
9. \(z.c_j = y.c_{j+1}\)
10. \(y.n = t - 1\)
11. for \(j = x.n + 1\) downto \(i + 1\)
12. \(x.c_{j+1} = x.c_j\)
13. \(x.c_{i+1} = z\)
14. for \(j = x.n\) downto \(i\)
15. \(x.key_{j+1} = x.key_j\)
16. \(x.key_i = y.key_t\)
17. \(x.n = x.n + 1\)
18. \(\text{DISK-WRITE}(y)\)
19. \(\text{DISK-WRITE}(z)\)
20. \(\text{DISK-WRITE}(x)\)
Inserting a key into a B-tree in a single pass down the tree

We insert a key $k$ into a B-tree $T$ of height $h$ in a single pass down the tree, requiring $O(h)$ disk accesses. The CPU time required is $O(t^h) = O(t \log_t n)$. The B-TREE-INSERT procedure uses B-TREE-SPLIT-CHILD to guarantee that the recursion never descends to a full node.

```
B-TREE-INSERT($T, k$)
1   $r = T.root$
2   if $r.n == 2t - 1$
3       $s = ALLOCATE-NODE()$
4       $T.root = s$
5       $s.leaf = FALSE$
6       $s.n = 0$
7       $s.c_1 = r$
8       B-TREE-SPLIT-CHILD($s, 1$)
9       B-TREE-INSERT-NONFULL($s, k$)
10  else B-TREE-INSERT-NONFULL($r, k$)
```

Lines 3–9 handle the case in which the root node $r$ is full: the root splits and a new node $s$ (having two children) becomes the root. Splitting the root is the only way to increase the height of a B-tree. The figure in the next page illustrates this case. Unlike a binary search tree, a B-tree increases in height at the top instead of at the bottom.
Various cases of inserting into a B-tree.
Deletion

Deletion from a B-tree is analogous to insertion but a little more complicated, because we can delete a key from any node, not just a leaf, and when we delete a key from an internal node, we will have to rearrange the node’s children.

As in insertion, we must guard against deletion producing a tree whose structure violates the B-tree properties.

Just as we had to ensure that a node didn’t get too big due to insertion, we must ensure that a node doesn’t get too small during deletion (except that the root is allowed to have fewer than the minimum number $t - 1$ of keys).

Just as a simple insertion algorithm might have to back up if a node on the path to where the key was to be inserted was full, a simple approach to deletion might have to back up if a node (other than the root) along the path to where the key is to be deleted has the minimum number of keys.

The procedure B-TREE-DELETE deletes the key $k$ from the subtree rooted at $x$. We design this procedure to guarantee that whenever it calls itself recursively on a node $x$, the number of keys in $x$ is at least the minimum degree $t$. Note that this condition requires one more key than the minimum required by the usual B-tree conditions, so that sometimes a key may have to be moved into a child node before recursion descends to that child.
Deletion \((\text{cotd.})\)

We sketch how deletion works instead of presenting the pseudocode.

1. 1. If the key \(k\) is in node \(x\) and \(x\) is a leaf, delete the key \(k\) from \(x\).

2. 2. If the key \(k\) is in node \(x\) and \(x\) is an internal node, do the following:
   a) If the child \(y\) that precedes \(k\) in node \(x\) has at least \(t\) keys, then find the predecessor \(k_0\) of \(k\) in the subtree rooted at \(y\). Recursively delete \(k_0\), and replace \(k\) by \(k_0\) in \(x\). (We can find \(k_0\) 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 \(k_0\) of \(k\) in the subtree rooted at \(z\). Recursively delete \(k_0\), and replace \(k\) by \(k_0\) in \(x\). (We can find \(k_0\) and delete it in a single downward pass.)
   c) Otherwise, 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 \(y\) now contains \(2t-1\) keys. Then free \(z\) and recursively delete \(k\) from \(y\).

3. 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 \(x\).
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 $x.c_i$ 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.
Example

Deleting keys from a B-tree.
The minimum degree for this B-tree is $t = 3$, so a node (other than the root) cannot have fewer than 2 keys. Nodes that are modified are lightly shaded.

a) The B-tree of our previous example.
b) (b) Deletion of F. This is case 1: simple deletion from a leaf.
c) Deletion of M. This is case 2a: the predecessor L of M moves up to take M’s position.
d) Deletion of G. This is case 2c: we push G down to make node DEGJK and then delete G from this leaf (case 1).
e) Deletion of D. This is case 3b: the recursion cannot descend to node CL because it has only 2 keys, so we push P down and merge it with CL and TX to form CLPTX; then we delete D from a leaf (case 1). (e1) After (e), we delete the root, and the tree shrinks in height by one.

f) Deletion of B. This is case 3a: C moves to fill B’s position and E moves to fill C’s position.