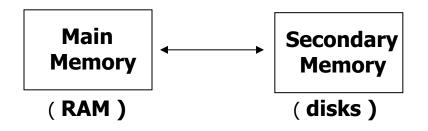
B-Trees

- Large degree B-trees used to represent very large dictionaries that reside on disk.
- Smaller degree B-trees used for internalmemory dictionaries to overcome cache-miss penalties.

B-Trees



x a pointer to some object

DISK - READ(x)

operations that access and/or modify the fields of x

DISK - WRITE(x)

others operations that access but do not \neg odify the fields of x

AVL Trees

- $n = 2^{30} = 10^9$ (approx).
- 30 <= height <= 43.
- When the AVL tree resides on a disk, up to 43 disk access are made for a search.
- This takes up to (approx) 4 seconds.
- Not acceptable.

Red-Black Trees

- $n = 2^{30} = 10^9$ (approx).
- 30 <= height <= 60.
- When the red-black tree resides on a disk, up to 60 disk access are made for a search.
- This takes up to (approx) 6 seconds.
- Not acceptable.

A Disk Page

useless content

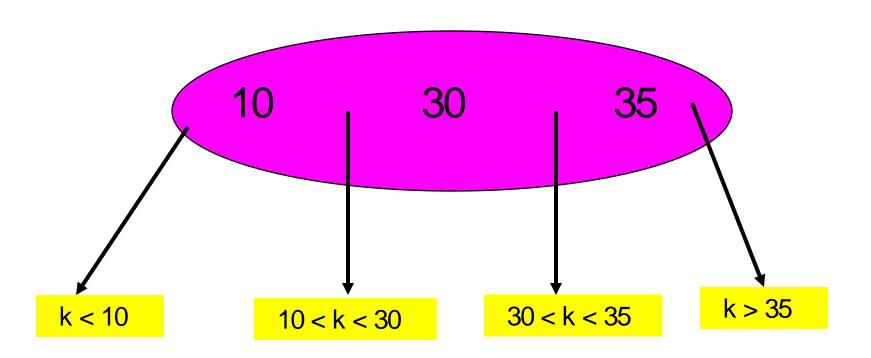
Useless content

A Search Tree Node

m-way Search Trees

- Each node has up to m 1 pairs and m children.
- m = 2 => binary search tree.

4-Way Search Tree



Maximum # Of Pairs

- Happens when all internal nodes are m-nodes.
- Full degree m tree.
- # of nodes = $1 + m + m^2 + m^3 + ... + m^{h-1}$ = $(m^h - 1)/(m - 1)$.
- Each node has m 1 pairs.
- So, # of pairs = $m^h 1$.

Capacity Of m-Way Search Tree

	2	200
3		* 10 - 1
	31	3.2 * 10 ¹¹ - 1

Definition Of B-

2-3 And 2-3-4 Trees

- B-tree of order m.
 - m-way search tree.
 - Not empty => root has at least 2 children.
 - Remaining internal nodes (if any) have at least ceil(m/2) children.
 - External (or failure) nodes on same level.

- 2-3 tree is B-tree of order 3.
- 2-3-4 tree is B-tree of order 4.

B-Trees Of Order 5 And 2

- B-tree of order m.
 - m-way search tree.
 - Not empty => root has at least 2 children.
 - Remaining internal nodes (if any) have at least ceil(m/2) children.
 - External (or failure) nodes on same level.

- B-tree of order 5 is 3-4-5 tree (root may be 2-node though).
- B-tree of order 2 is full binary tree.

Minimum # Of Pairs

- n = # of pairs.
- # of external nodes = n + 1.
- Height = h => external nodes on level h + 1.

Minimum # Of Pairs

$$n + 1 >= 2* ceil(m/2)^{h-1}, h >= 1$$

• m = 200.

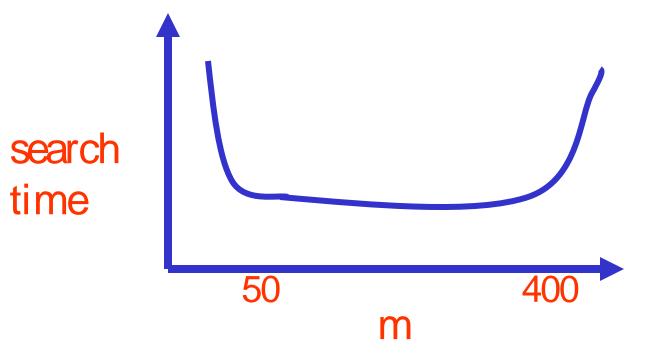
```
height # of pairs

2 >= 199
3 >= 19,999
4 >= 2 * 10^{6}-1
5 >= 2 * 10^{8}-1
```

$$h \le \log_{ceil(m/2)}[(n+1)/2] + 1$$

Choice Of m

- Worst-case search time.
 - (time to fetch a node + time to search node) * height



- convention :
 - Root of the B-tree is always in main memory.
 - Any nodes that are passed as parameters must already have had a DISK_READ operation performed on them.
- Operations:
 - Searching a B-Tree.
 - Creating an empty B-tree.
 - Splitting a node in a B-tree.
 - Inserting a key into a B-tree.
 - Deleting a key from a B-tree.

Node Structure

 $n c_0 k_1 c_1 k_2 c_2 \dots k_n c_n$

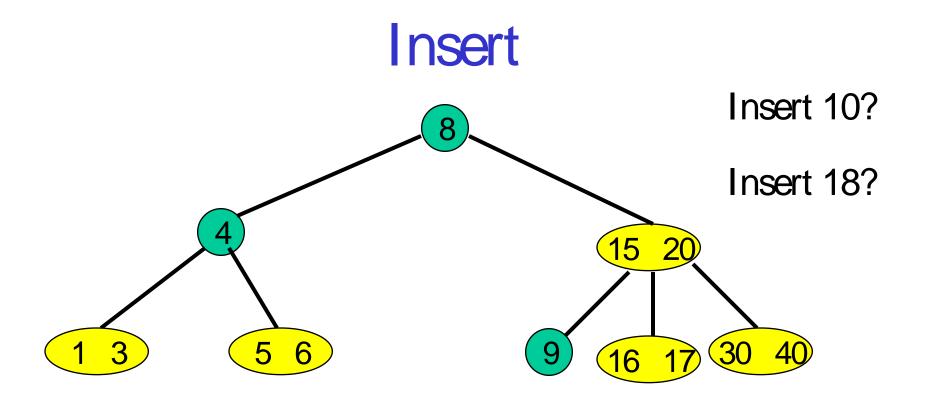
- c_i is a pointer to a subtree.
- k_i is a dictionary pair(KEY).

Search

```
BT_Search(x, k)
   ← ()
  while < and > _{+1}[ ]
       do ← +1
      < and = _{+1}[ ]
     then return(, +1)
         [ ] then return NULL
            else DISK-READ(C[])
                 return B-Tree-Search(C[],)
```

B-Tree-Created(T) : Algorithm: **B-Tree-Create(T)** $\{ x \leftarrow Allocate - Node() \}$ Leaf[x] \leftarrow TRUE $n[x] \leftarrow 0$ DISK - WRITE(x) $root[T] \leftarrow x$

time: O(1)



Insertion into a full leaf triggers bottom-up node splitting pass.

Split An Overfull Node

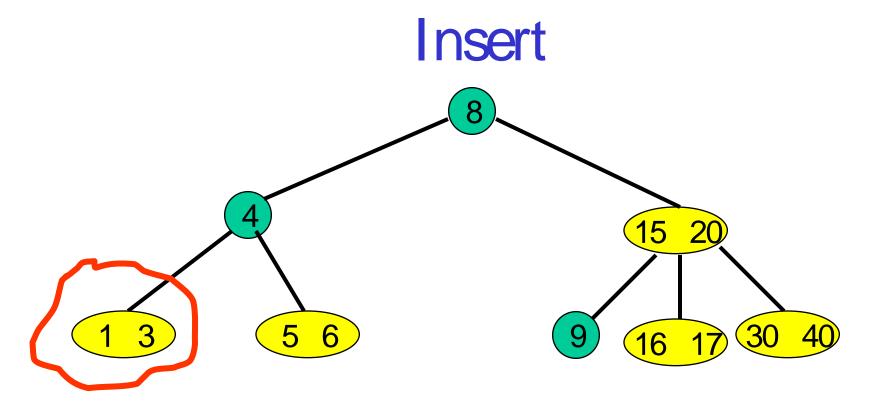
 $m c_0 k_1 c_1 k_2 c_2 \dots k_m c_m$

- c_i is a pointer to a subtree.
- k_i is a dictionary pair(KEY).

 $ceil(m/2)-1 c_0 k_1 c_1 k_2 c_2 ... k_{ceil(m/2)-1} c_{ceil(m/2)-1}$

m-ceil(m/2) $c_{ceil(m/2)} k_{ceil(m/2)+1} c_{ceil(m/2)+1} ... k_m c_m$

• $k_{\text{ceil}(m/2)}$ plus pointer to new node is inserted in parent.



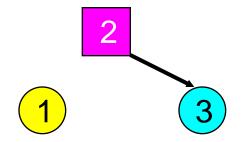
- Insert a pair with key = 2.
- New pair goes into a 3-node.

Insert Into A Leaf 3-node

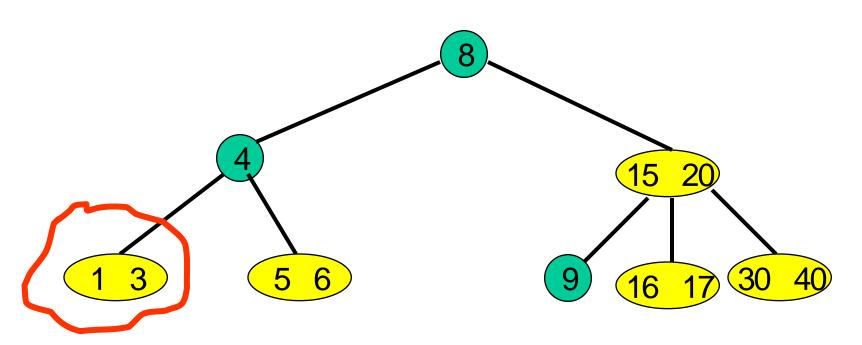
 Insert new pair so that the 3 keys are in ascending order.



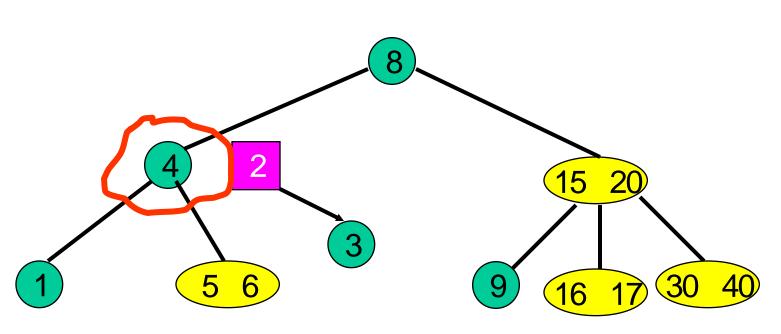
Split overflowed node around middle key.



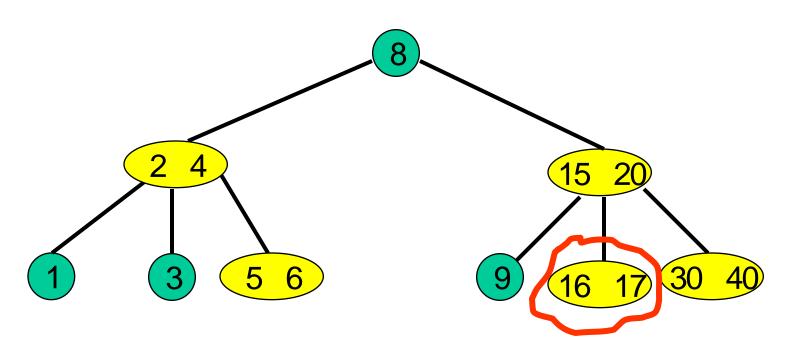
 Insert middle key and pointer to new node into parent.



• Insert a pair with key = 2.



• Insert a pair with key = 2 plus a pointer into parent.



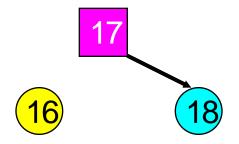
Now, insert a pair with key = 18.

Insert Into A Leaf 3-node

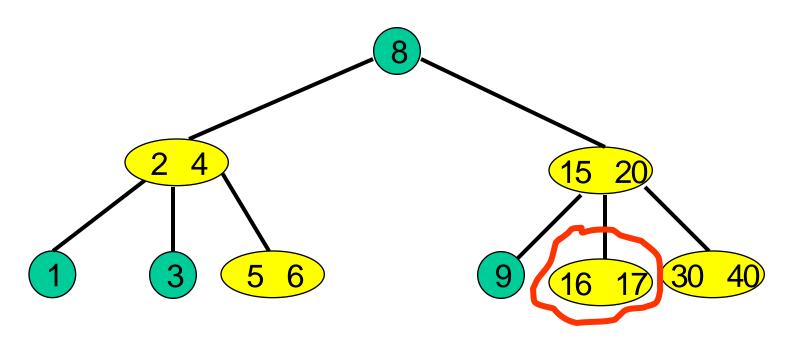
 Insert new pair so that the 3 keys are in ascending order.



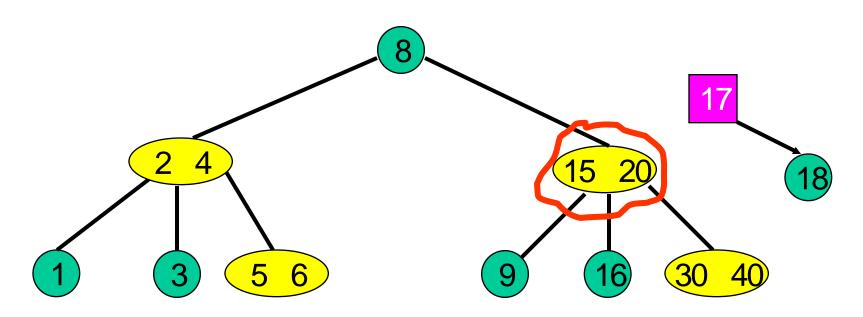
Split the overflowed node.



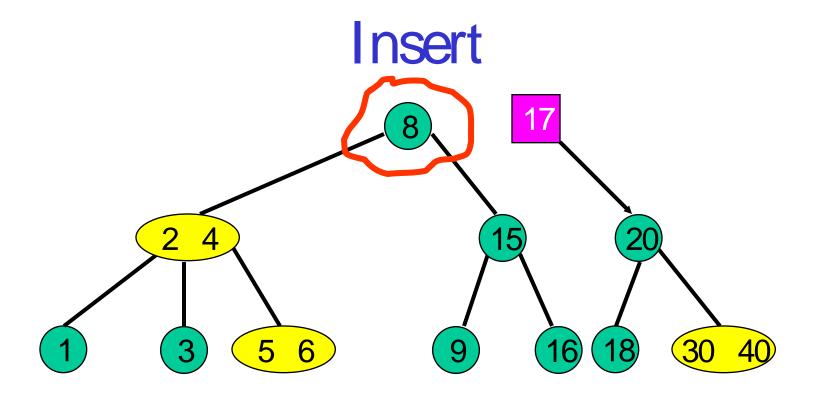
 Insert middle key and pointer to new node into parent.



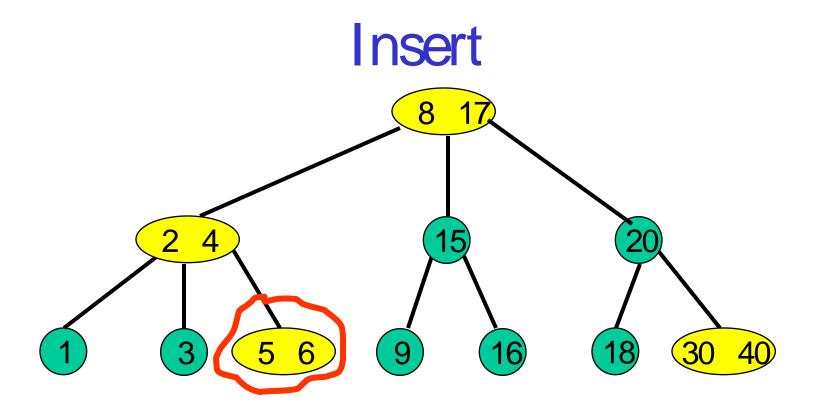
Insert a pair with key = 18.



• Insert a pair with key = 17 plus a pointer into parent.



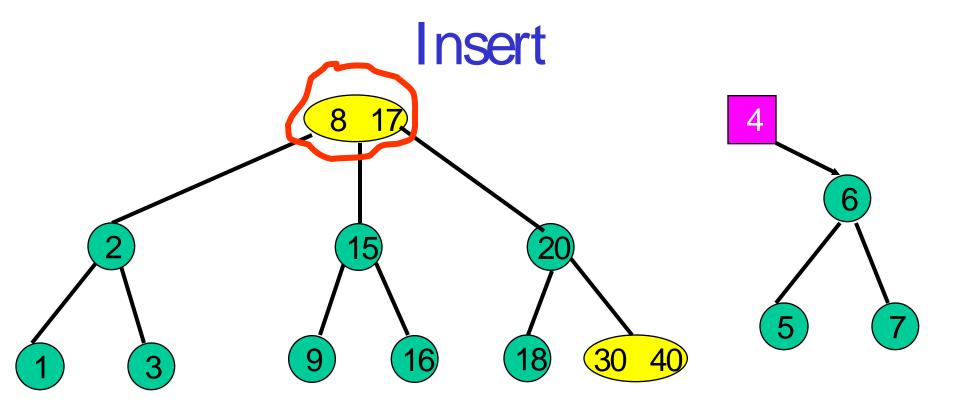
• Insert a pair with key = 17 plus a pointer into parent.



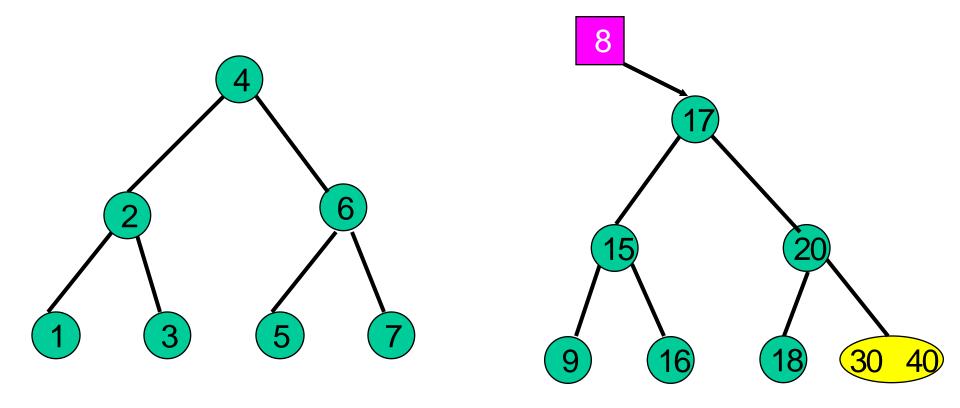
Now, insert a pair with key = 7.

Insert 6 (9)

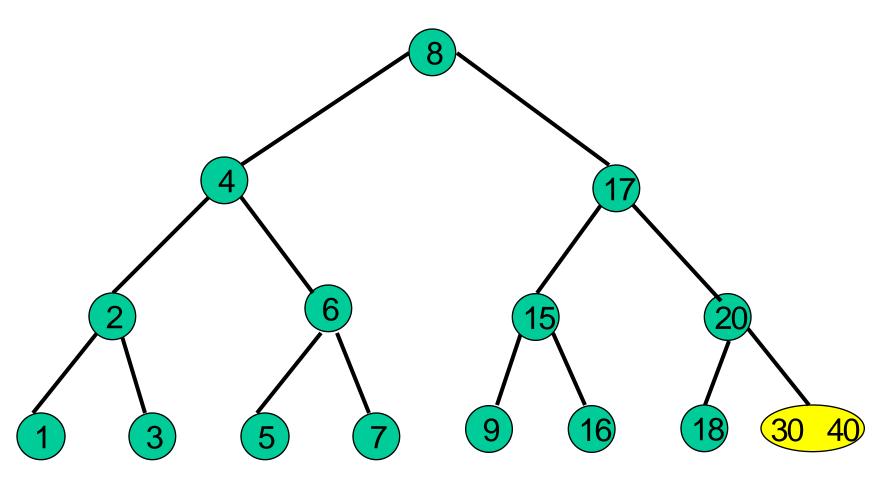
• Insert a pair with key = 6 plus a pointer into parent.



• Insert a pair with key = 4 plus a pointer into parent.



- Insert a pair with key = 8 plus a pointer into parent.
- There is no parent. So, create a new root.



• Height increases by 1.

Btree::InsertNode(Key k, Element e){bool overflow = Insert(root, k, e);

```
    Bool Insert(node* x, Key k, Element e)

      if (leaf(x))
            insertLeaf(x, k, e);
            if (size(x) > m-1) return true;
            else return false;
      idx = keySearch(x, k);
      bool overflow = Insert(x->C[idx], k, e);
```

```
if (overflow)
   <Key, Node* > newpair = split(x->C[idx]);
   InsertPair(x, newpair);
   if(size(x) > m-1)
     return true;
   else return false;
```

• Exercises: P609-3