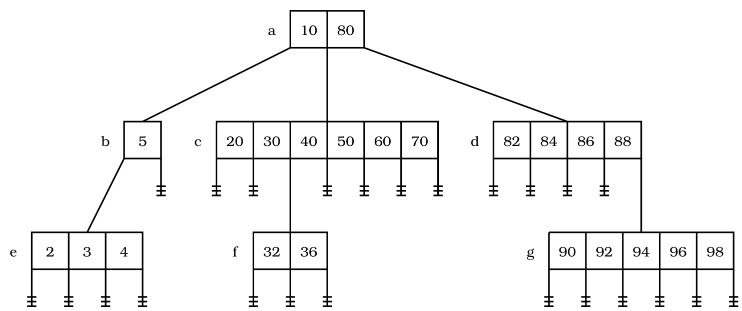
m-way Search Trees

m-way Search Tree: Empty, or if not empty then:

- Each internal node has q children and $q-1$ elements, for $2 \leq q \leq m$.
- Nodes with p elements have exactly $p+1$ children.
- Suppose a node has p elements. Let k_1, k_2, ... k_p be the keys of these elements. Then $k_1 < k_2 < ... < k_p$. Let $c_0, c_1, ..., c_p$ be the $p+1$ children of the node.
  - The elements in the subtree with root c_0 have keys smaller than k_1.
  - The elements in the subtree rooted at c_i have keys larger than k_i and smaller than k_{i+1}, $1 \leq i < p$.
  - The elements in the subtree rooted at c_p have keys larger than k_p.

Example



- a: 2,b,(10,c),(80,d)
- b: 1,e,(5,0)
- c: 6,0,(20,0),(30,f),(40,0),(50,0),(60,0),(70,0)
- d: 4,0,(82,0),(84,0),(86,0),(88,g)
- e: 3,0,(2,0),(3,0),(4,0)
- f: 2,0,(32,0),(36,0)
- g: 5,0,(90,0),(92,0),(94,0),(96,0),(98,0)
- Searching; Inserting (31,65);
- Deleting (20,84,5,10);
- Format $(n, c_0, (e_1, c_1), (e_2, c_2), \ldots, (e_n, c_n))$

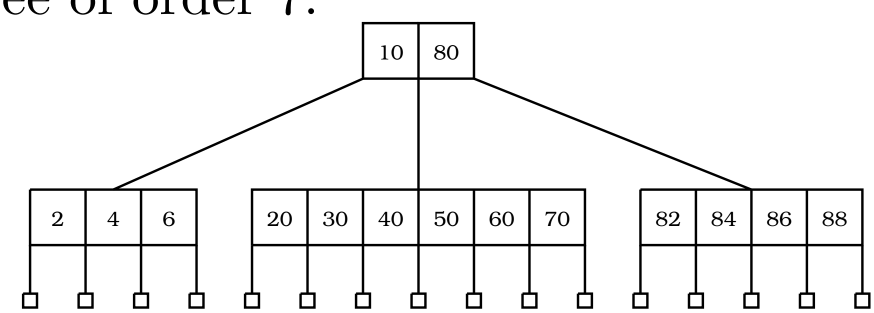
B-Trees of Order $m > 2$

(Different from textbook [W])

A B-Tree of order m is an m-way search tree. If the B-tree is not empty, the corresponding extended tree satisfies the following properties:

- The root has at least two children.
- All internal nodes other than the root have at least $\lceil m/2 \rceil$ children.
- All external nodes are at the same level.

A B-tree of order 7.



Properties

- $m > 2$ because they cannot represent all possible sets.
- B-Tree of order 3 is a 2-3 tree.
- B-Tree of order 4 is a 2-3-4 tree (Same as RB-Tree).

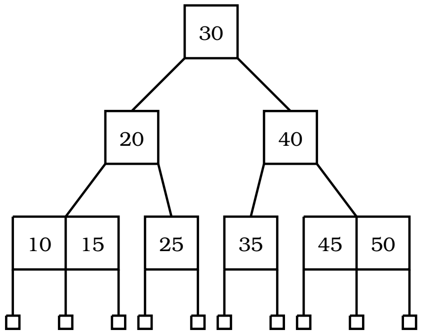
Lemma 11.3: Let T be a B-tree of order m and height h. Let $d = \lceil m/2 \rceil$ and let n be the number of elements in T.

1. $2d^{h-1} - 1 \leq n \leq m^h - 1$
2. $log_m(n+1) \leq h \leq log_d(\frac{n+1}{2}) + 1.$

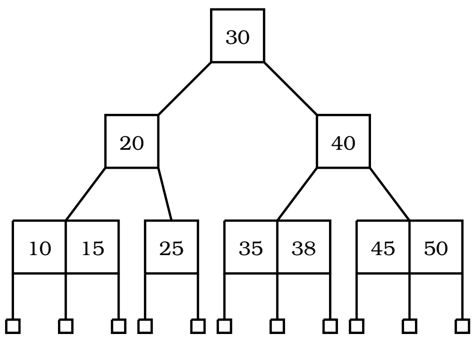
Proof: (1) $\Rightarrow$ (2). (1) follows from the fact that the minimum number of nodes on levels 1, 2, 3, 4, ..., h is 1, 2, $2d$, $2d^2$, ..., $2d^{h-2}$, and the maximum num. is 1, m, m^2, ..., m^{h-1}. The number of null pointers $= n + 1$.

- A B-tree of order 200 and height 3 has at least 19,999 elements and therefore can represent all UCSB students.
- A B-tree of order 200 and height 5 has at least 199,999,999 and therefore can represent all U.S. voters.
- The order of a B-Tree is determined by the disk block size and size of individual elements.
- For obvious reasons all the B-tree examples have small order.
- Searching is like in an m-way search tree.

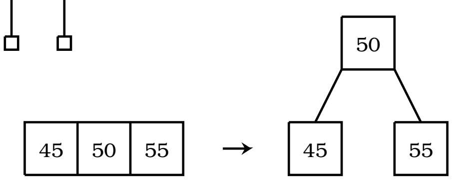
Insert Example (B-Tree of Order 3)

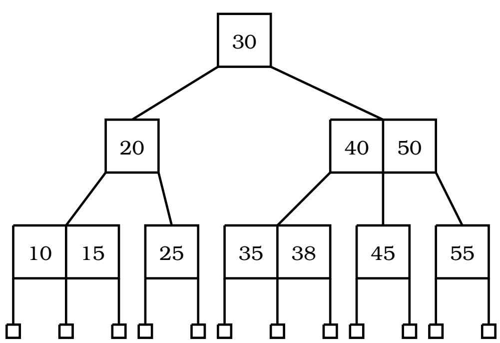


I 38

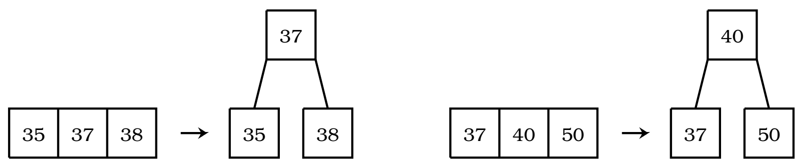


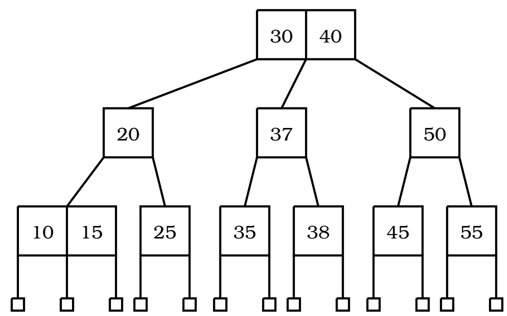
I 55



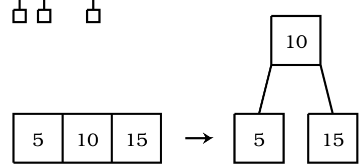


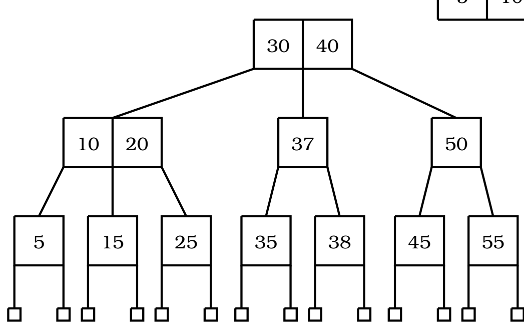
I 37


35 37 38
37
35 38
37 40 50
40
37 50

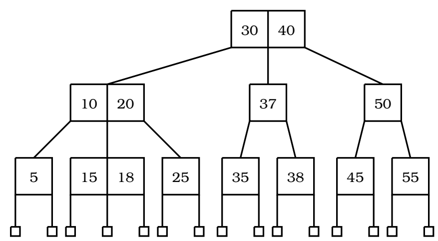

30 40
20 37 50
10 15 25 35 38 45 55

I 5

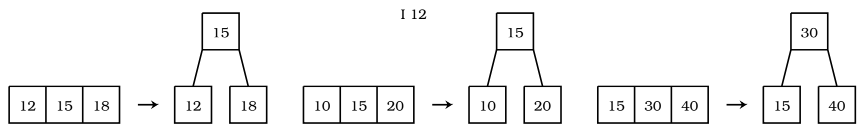

5 10 15
10
5 15

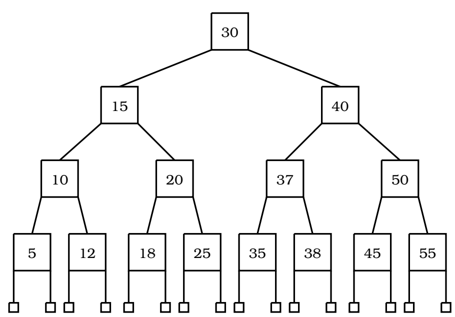

30 40
10 20 37 50
5 15 25 35 38 45 55

I 18


30 40
10 20
37
50
5
15 18
25
35
38
45
55

I 12


12 15 18
15
12
18
10 15 20
15
10
20
15 30 40
30
15
40

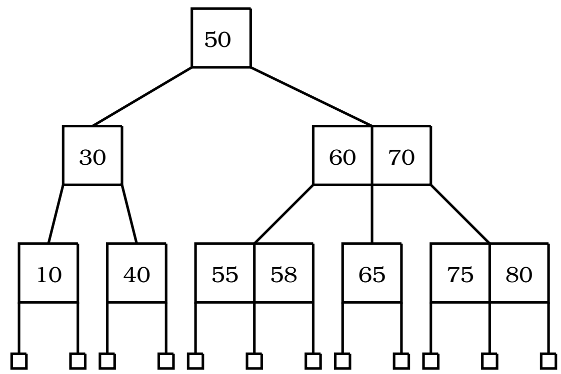

30
15
40
10
20
37
50
5
12
18
25
35
38
45
55

Insertion

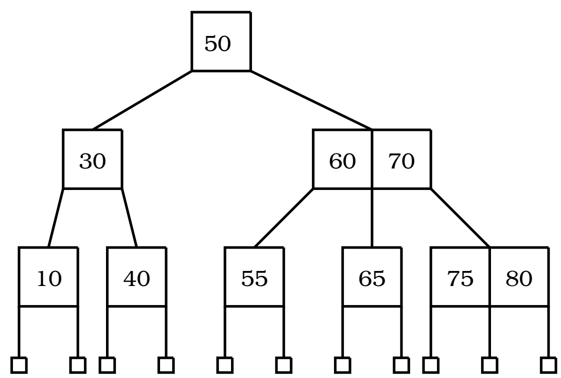
```
Nodes are of the form n,c_0,(e_1,c_1),...,(e_n,c_n), where
   the e's are the values or keys and the c's are the pointers.

Procedure Insert(t,e) {// t points to root, and e will be inserted
   c = NULL;  // (e,c) is to be inserted in leaf node
   Search(t,e,P,found); // returns found=true if e in the tree
       // and P will point to the node in main memory that has e;
       // Returns false if e is not in the B-tree and P will point
       // to the last node visited (leaf node) during the search;
   Done = false;
   if not found {
      while P != NULL & not Done do
        {Insert (c,e) into appropriate position in node P;
         Let the resulting node be P -> n,c_0,(e_1,c_1),...,(e_n,c_n)
```

```
        if P->n <= m-1 {Output P to Disk; Done = true;}
          else { e = P->e_{ceil(m/2)};
                 d = ceil(m/2);
                 Split P into two nodes (in main memory)
                   P: d-1,c_0,(e_1,c_1),...,(e_{d-1},c_{d-1})
                   Q: m-d,c_d,(e_{d+1},c_{d+1}),...,(e_m,c_m)
                   Output P and Q to Disk;
                   c = Q;
                   P = Parent(P); // Parent may be obtained from a
                      // stack that is built by the Search procedure;
                }
       }
      if not Done { Create new node Q in memory;
                      Q: 1,t,(e,c);
                    t = Q;
                    Output t to Disk;
                  }
}   }
```

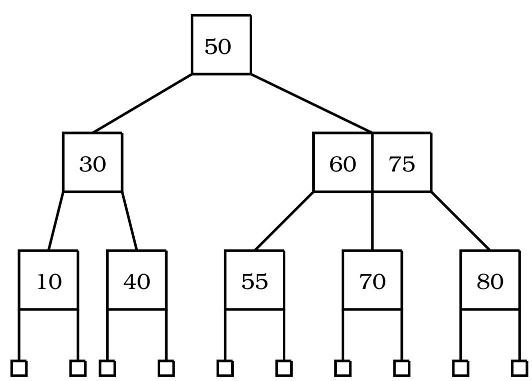
Delete Example



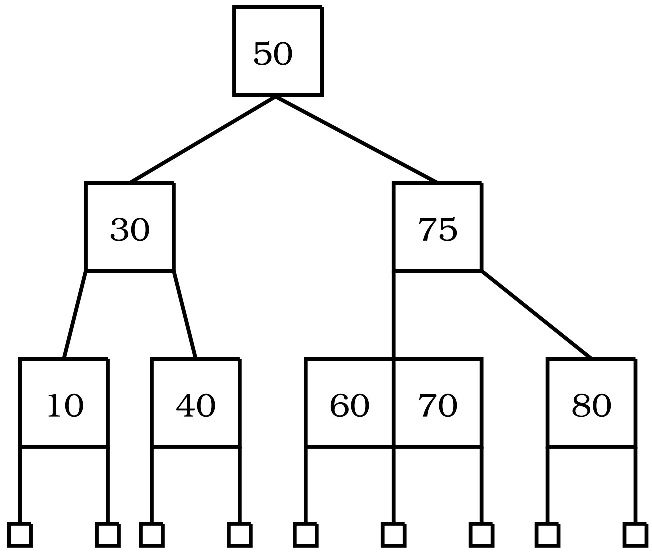
D 58



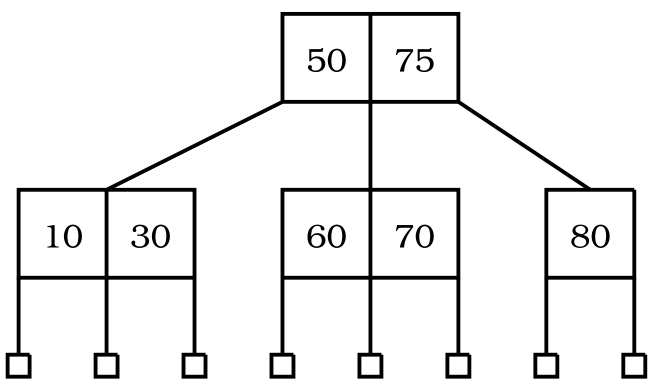
D 65



D 55



D 40



Deletion

Nodes are of the form n,c_0,(e_1,c_1),...,(e_n,c_n), where the e's are the values or keys and the c's are the pointers.

```
Procedure Delete(t,e) {

   // t points to root, and e will be deleted

   Search(t,x,P,found); // returns found=true if e in the tree
       // and P will point to the node in main memory that has e;
       // Returns false if e is not in the B-tree and P will point
       // to the last node visited (leaf node) during the search;
```

```
if found {
   Let P point to node n,c_0,(e_1,c_1),...,(e_n,c_n),
         and e_i  has value e;
   if P->c_0 != 0 // P is not a leaf node
     { Q = P->c_i;  // Reads from Disk P->c_i and
                    // stores it in memory node Q
       While Q is not a leaf node do
         Q = Q->c_0;  //Reads from Disk P->c_i and
                    // stores it in memory node Q
       P->e_i = Q->e_1
       Write P on Disk;
       P = Q;
       i = 1;
     }
```

```
delete (P->e_i, P->c_i) from
        P: n,c_0,(e_1,c_1),...,(e_n,c_n)
        and replace P->n by P->n-1;
while (P->n < Ceil(m/2)-1) && (P != t) do
  { if P has a nearest right sibling Y
    {Let Z point to the parent of P and Y;
     Let j be such that Z->c_{j-1} == P && Z->c_j == Y;
     if Y->n >= ceil(m/2)
       { // can borrow from right sibling
         P->e_{P->n+1} = Z->e_j; //move from Z to P
         P->c_{P->n+1} = Y->c_0;
         P->n = P->n+1;
         Z->e_j = Y->e_1; //move e_1 from Y to Z
         Y->(n,c_0,(e_1,c_1),... ) =>
                Y->(n-1,c_1,(e_2,c_2),... ); //e_1 is deleted
         Output nodes P, Z & Y on Disk;
         return;
```

```
            }

         //Has a right child but cannot borrow from it
         r = 2 Ceil(m/2)-2;
         // Borrow from parent and combine P and Y into one node
         Output ( r, P->c_0,(P->e_1,P->c_1),...,
                          (P->e_{P->n},P->c_{P->n}),
                    (Z->e_j,Y->c_0),
                    (Y->e_1,Y->c_1),...,
                          (Y->e_{Y->n},Y->c_{Y->n}) )
                as new node P;
         Node P is now node Z except that (Z->e_j,Z->c_j) is deleted;
        }
        else {do the nearest left sibling instead}
       }
 if P->n != 0 {Output P onto Disk;}
         else { t = P->c_0; }
}
```

```
}
```

Extensions

- Above material from Horowitz and Sahni Fundamentals of DS (CS Press). But the algorithms were modified by Prof. Gonzalez to be more OO.

- A B'-Tree is like the B-Tree, but the values are at the failure nodes (instead of a null pointer we have a pointer to the data). Internal nodes have keys to direct the search. The Textbook [W] covers B'-Trees,but calls them B-Trees.

- B^*-Tree: The root has at least two children and at most $2\lceil(2m-2)/3\rceil+1$. Internal nodes have at least $\lceil(2m-2)/3\rceil$ and at most m children. (Saves space).