Chapter 11
Multiway Search Trees

11.1 m-Way Search trees

11.1.1 Definition of m-Way Search Trees

If AVL tree were used to represent a very large collection of elements, it would be on disk. To search among n keys requires $h=1.44 \log_2 (n+1)$ disk accesses in the worst case.

If $n=10^6$, $1.44\log_2 (n+1) \approx 28$ --- too bad!

Recall that the block of a disk access (I/O) is much larger than the node of a binary tree. If we use AVL tree for index, accessing a node is actually accessing a block of which most part are useless, as shown below:

an AVL node in a disk block

To break the $\log_2(n+1)$ barrier on tree height resulting from the use of binary search trees, we must use search trees whose degree is more than 2.

In practice, we use the largest degree for which the tree node fits into a block.

Definition: An m-way search tree, either is empty or satisfies the following properties:

(1) The root has at most m subtrees and has the following structure:

$$n, A_0, (E_1, A_1), (E_2, A_2), \ldots, (E_n, A_n)$$

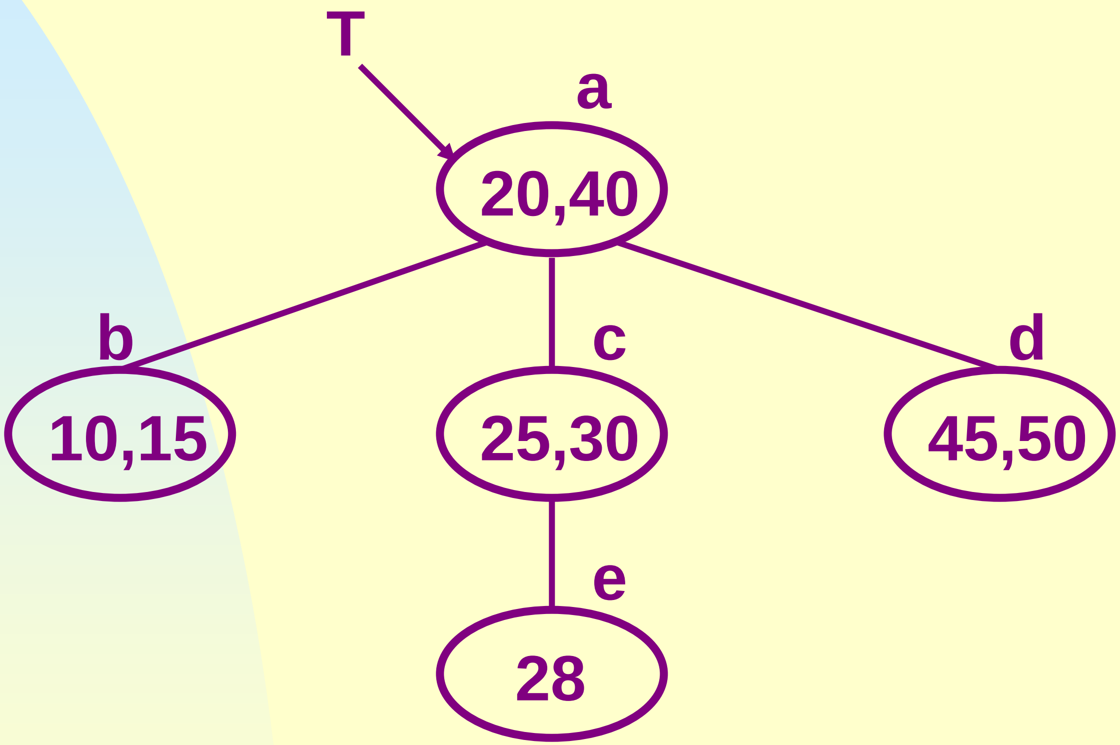
where the A_i , $0 \le i \le n < m$, are pointers to subtrees, and E_i, $1 \le i \le n < m$, are elements. Each E_i has a key $E_i.K$.

(2) $E_i.K < E_{i+1}.K$, $1 \le i < n$.

(3) Let $E_0.K = -\infty$ and $E_{n+1}.K = \infty$. All key values in subtree A_i are less than $E_{i+1}.K$ and greater than $E_i.K$, $0 \le i \le n$.

(4) The subtrees A_i , $0 \le i \le n$, are also m-way search trees.

The following is a 3-way search tree:



In a tree of degree m and height h, the maximum number of nodes is

$$\sum_{0 \le i \le h-1} m^i = (m^h - 1)/(m - 1)$$

Each node has at most m-1 keys, the maximum number of keys is m^h-1.

- **for a binary tree with h=3, it is 7.**
- **for a 200-way tree with h=3, it is 8*10^6-1.**

To achieve a performance close to that of the best m-way search tree for a given number of elements n, the search tree must be balanced.

11.1.2 Searching an m-way Search Tree

The searching is easy, and the next slide gives a high level description of the algorithm to search an m-way search tree.

```
// Search an m-way search tree for an element with key x.
// Return the element if found, else return NULL.
E0.K = -MAXKEY;
for ( *p=root; p; p= Ai)
{
    Let node p have the format n, A0, (E1, A1), …, (En, An);
    En+1.K = MAXKEY;
    Determine i such that Ei.K <= x < Ei+1.K;
    if (x==Ei.K) return Ei;
}
// x is not in the tree
return NULL;
```

Exercises: P609-3

11.2 B-Trees

11.2.1 Definition and Properties

A particular balanced m-way search tree is B-tree.

To define it, an external (or failure) node is added wherever we otherwise have a NULL pointer.

An external node represents a node that can be reached during a search only if the element being searched for is not in the tree.

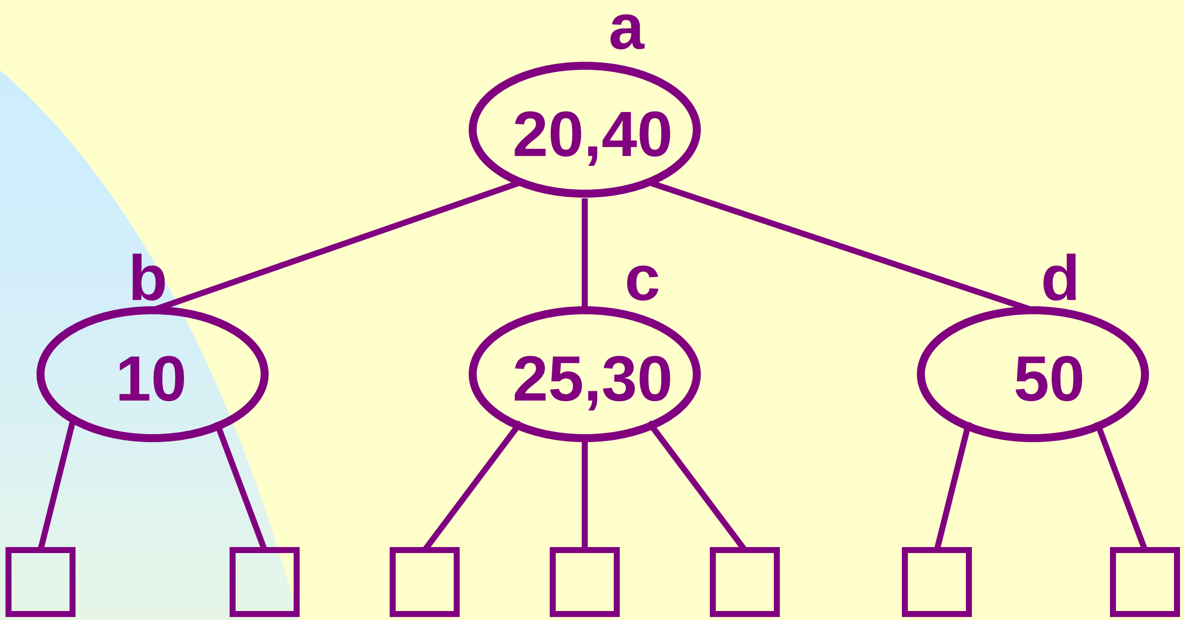
Definition: A B-tree of order m is an m-way search tree that is empty or satisfies the following properties:

(1) The root has at least two children.

(2) All nodes other than the root node and external nodes have at least $\lceil m/2 \rceil$ children.

(3) All external nodes are at the same level.

When m=3, all internal nodes have a degree of either 2 or 3, and a B-tree of order 3 is known as an 2-3 tree.



A B-tree of order 3

B-trees of order 2 are full binary trees.

For any $n \geq 0$ and $m > 2$, there is a B-tree of order m that contains n elements.

11.2.2 Number of Elements in a B-tree

Let t be a B-tree of order m in which all external nodes are at level l+1, and let N the number of keys in t. Then

(1) $N \le m^l - 1$ (upper bound).

(2) $N \ge 2\lceil m/2 \rceil^{l-1} - 1$ (lower bound).

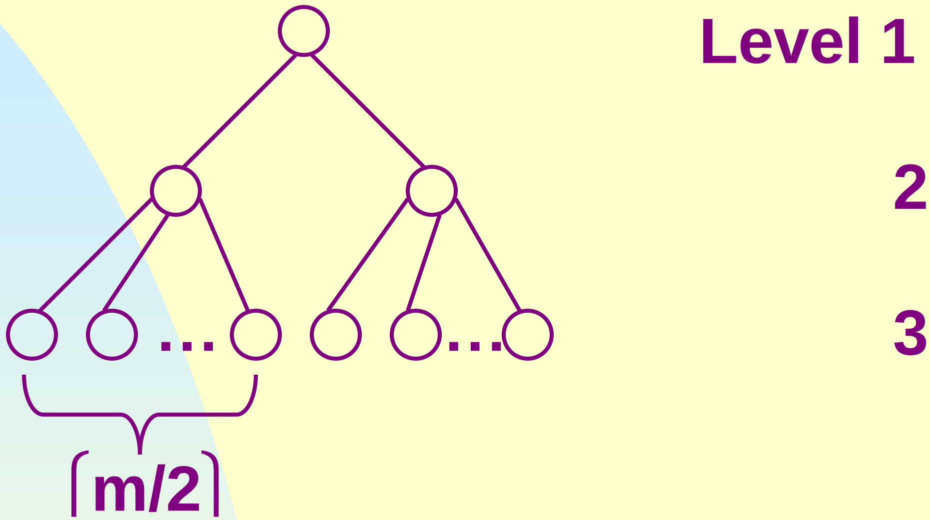
To show this, observe:

If l>1, there are at least 2 nodes at level 2,

there are at least 2 $\lceil m/2 \rceil$ nodes at level 3,

…

there are at least $2\lceil m/2 \rceil^{l-2}$ nodes at level l.



If the key values in the tree are $K_1, K_2, \ldots, K_N$, $K_i<K_{i+1}$, $1 \le i<N$, then the number of external nodes is N+1 because failures occur for $K_i<x<K_{i+1}$, $0 \le i \le N$,

where $K_0 = -\infty$ and $K_{N+1} = +\infty$.

Therefore,

$N+1 =$ number of failure nodes in t

$=$ number of nodes at level $l+1$

$\geq 2\lceil m/2 \rceil^{l-1}$

$N \geq 2\lceil m/2 \rceil^{l-1} - 1.$

$l \leq \log_{\lceil m/2 \rceil}((N+1)/2)+1.$

In worst case to search the B-tree need l accesses.

Assume m=200, and note l is integer:

(1) For $N \le 2*10^6-2$,

$l \le \lfloor \log_{100}(10^6-1/2) \rfloor +1=3.$

(2) For $N \le 2*10^8-2$,

$l \le \lfloor \log_{100}(10^8-1/2) \rfloor +1=4.$

11.2.3 Insertion into a B-Tree

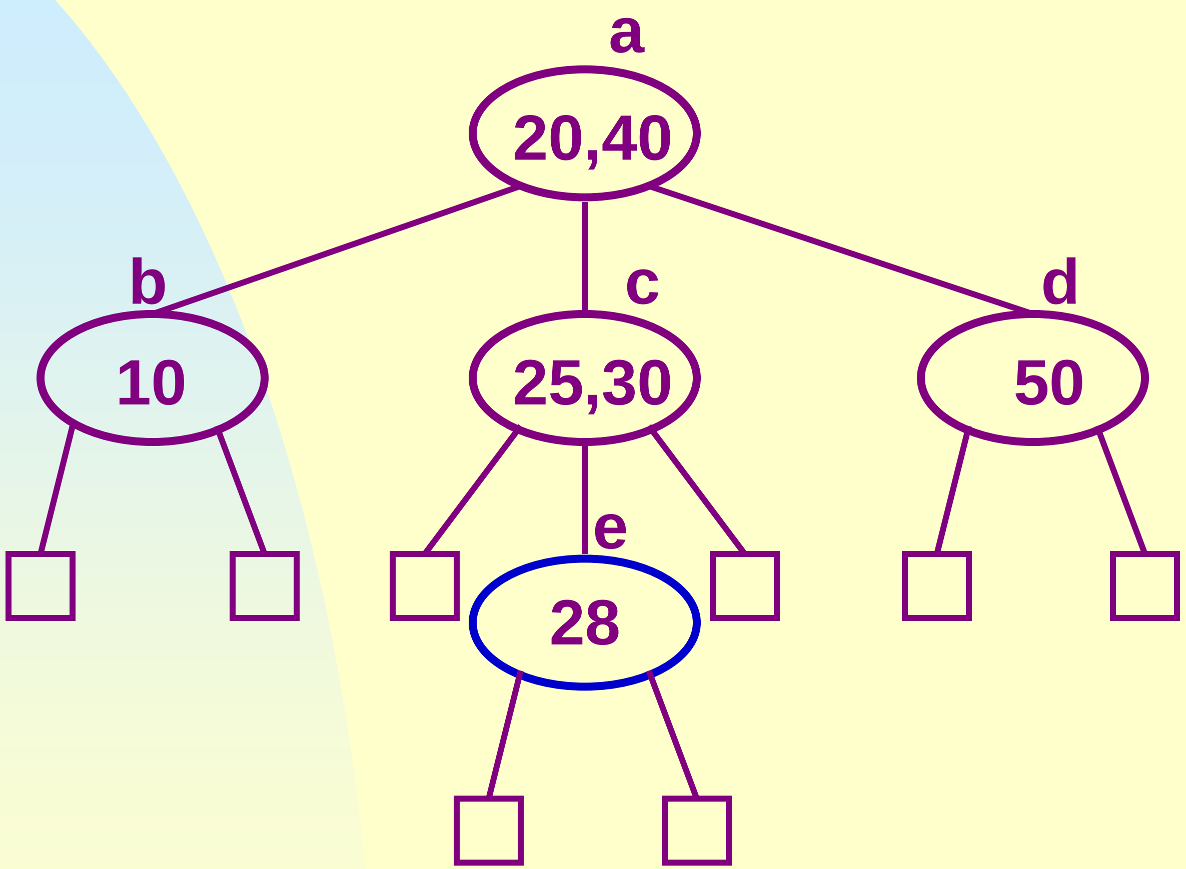
Note that we can't insert a new node in the position of an external node because:

1 One key might be too few;

2 External nodes must be at the same level.

As shown in the next slide.

We can't insert 28 into a B-Tree like this:



Basic ideas of the insertion algorithm for B-tree of order m:

(1) Perform a search to determine the leaf node, p, into which the new key is to be inserted.

(2) If the insertion results in p having m keys, the node p is split. Otherwise, the new p is written to the disk and the insertion is complete.

To split the node, assume that following the insertion, p has the format

$m, A_0, (E_1, A_1), \ldots, (E_m, A_m)$ and $E_i.K < E_{i+1}.K$, $1 \le i < m$.

The node is split into 2 nodes, p and q, with the following formats:

node p: $\lceil m/2 \rceil -1, A_0, (E_1, A_1), \ldots, (E_{\lceil m/2 \rceil -1}, A_{\lceil m/2 \rceil -1})$ (11.5)

node q: $m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, (E_{\lceil m/2 \rceil +1}, A_{\lceil m/2 \rceil +1}), \ldots, (E_m, A_m)$

And the tuple $(E_{\lceil m/2 \rceil}, q)$ is to be inserted into the parent of p.

Inserting into the parent may require us to split the parent, and the splitting process can propagate all the way up to the root.

When the root split, a new root with a single key is created, and the height of the B-tree increases by one.

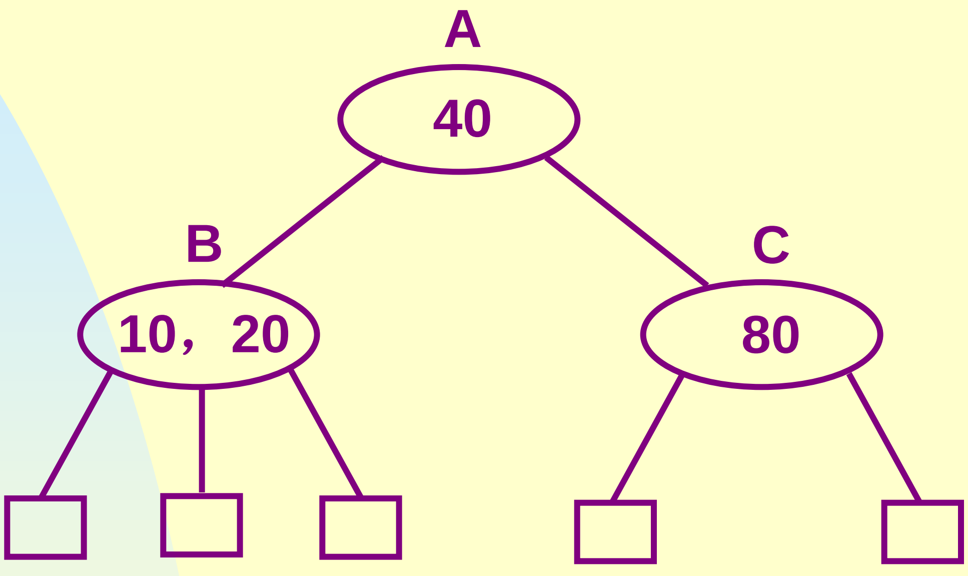
Note that when m=2, $\lceil m/2 \rceil$-1=0, this means the above method does not work for m=2.

The next slide gives a high-level description of the insertion algorithm for a disk resident B-tree.

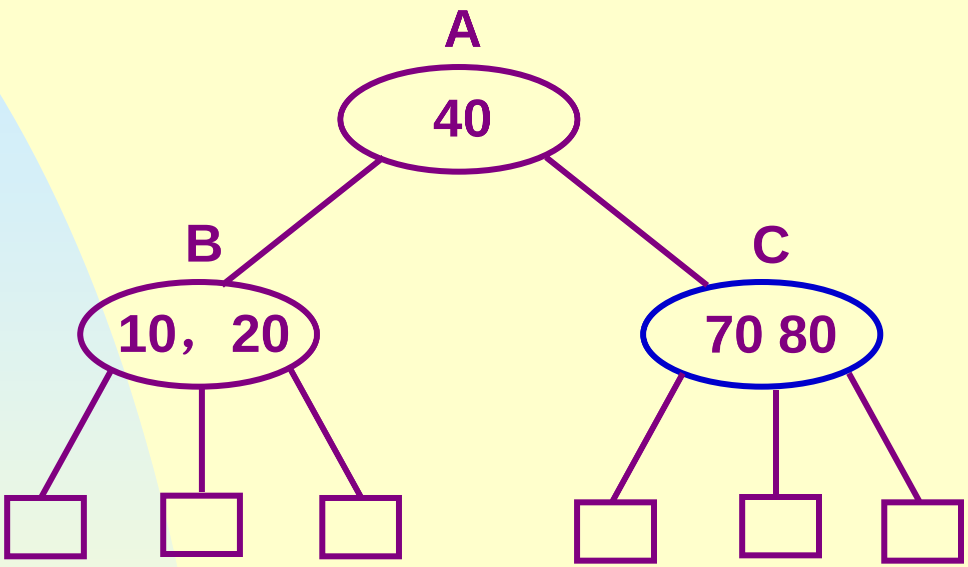
```
// Insert element x into a disk resident B-tree.
Search the B-tree for an element E with key x.K;
if such an E is found, replace E with x and return;
else let p be the leaf into which x is to be inserted;
q = NULL;
for (e=x; p; p=p→parent())   // the parent of root is NULL
{ // (e, q) is to be inserted into p
    Insert (e, q) into appropriate position in node p;
    Let the resulting node have the form:
                                          n, A0, (E1, A1), …, (En, An);
    if (n<=m-1) {   // the resulting node is not too big
        write node p to disk; return;
    }
```

```
    //node p has to be split
    Let node p and q be defined as in (11.5);
    e=E⌈m/2⌉;
    write nodes p and q to the disk;
}
// a new root is to be created
Create a new node r with format:  1, root, (e, q);
// when the tree is empty, root=p=0
root=r;
write  root to disk;
```

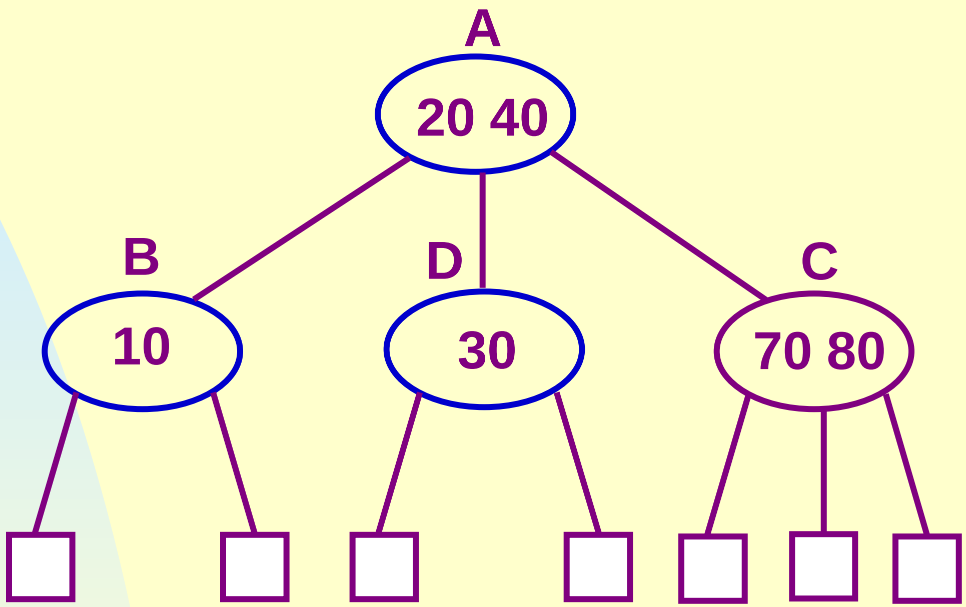
Example 11.1:



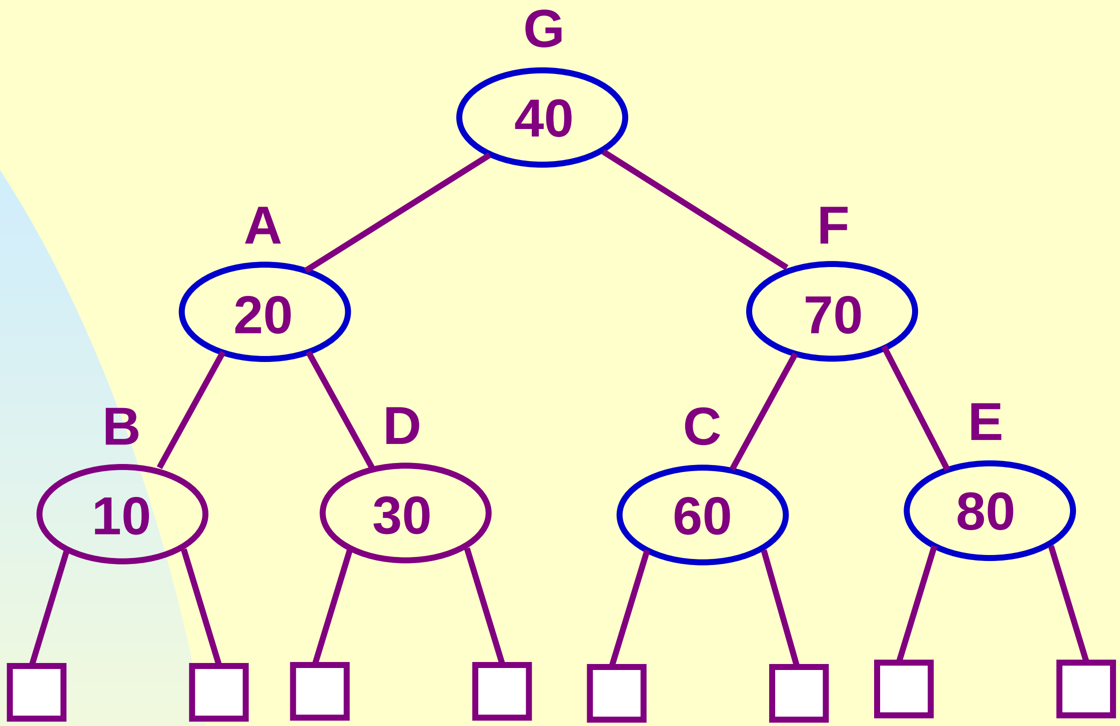
An initial 2-3 tree



70 inserted



30 inserted



60 inserted

Analysis of B-tree Insertion:

Let h be the height of the B-tree, then h disk accesses for the top-down search.

In the worst, all h of the accessed nodes may split during the bottom-up splitting pass. When a non-root node split, 2 nodes are written out. When the root split, 3 nodes are written out.

Assume that the h nodes read in during the top-down pass can be saved in memory so that they are not to be retrieved from disk during the bottom-up pass.

The total disk accesses is at most

$h+2(h-1)+3=3h+1$

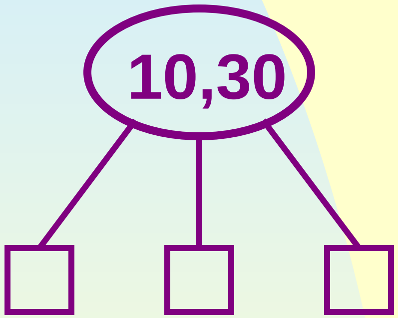
The average number of disk accesses is, however, approximately $h+1$ for lager m.

To show this, suppose we start with an empty B-tree and insert N elements into it.

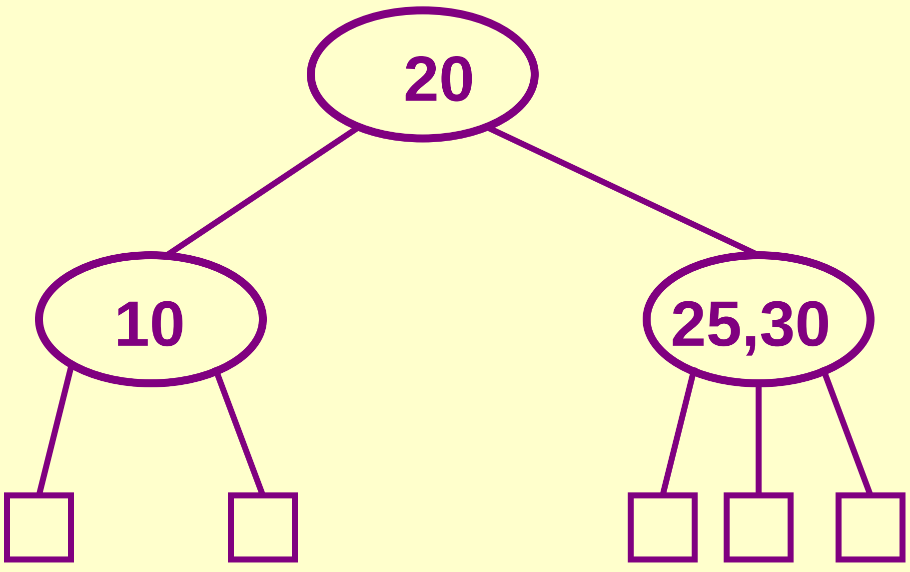
Let p be the number of internal nodes in the final B-tree with N elements, then the total number of nodes split is at most p-2. Because

- **Each time a node splits, at least one additional node is created.**
- **The splitting of the root node create two additional nodes and the root is the first to split.**
- **The first node created from no splitting.**

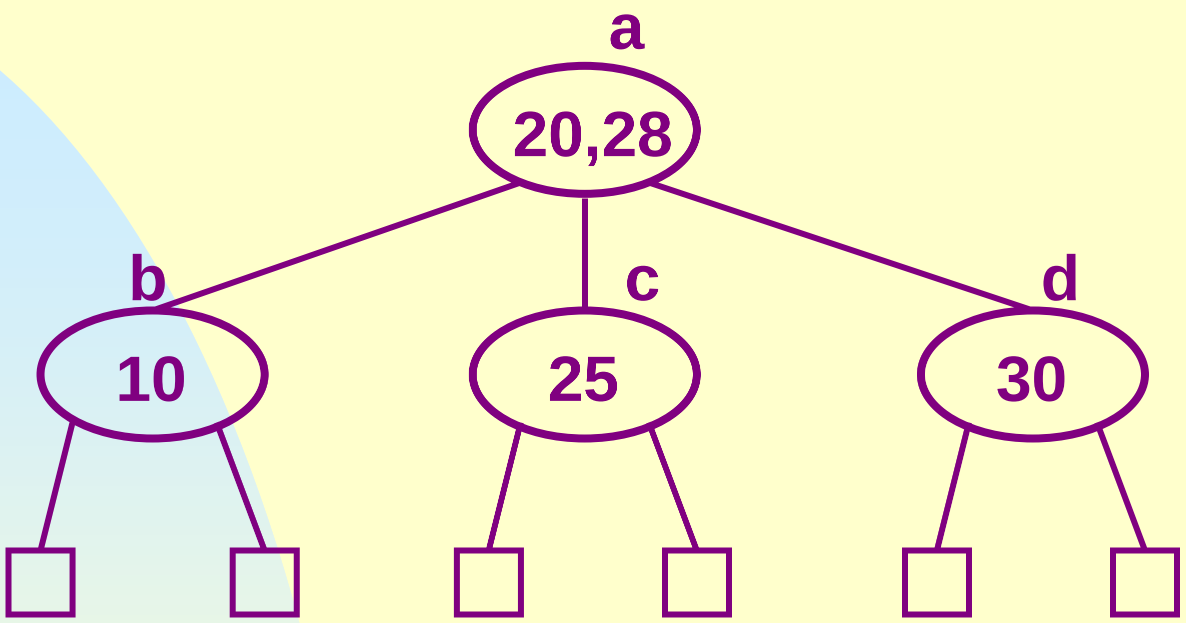
As shown in the following:



(a) p=1,s=0



(b) p=3,s=1



(c) p=4, s=2

A B-tree of order m with p nodes has at least

$$1 + (\lceil m/2 \rceil - 1)(p-1) \quad \text{elements.}$$

(root) (other nodes)

The average number of splitting s_{avg}

s_{avg} = (total number of splits)/N

$\le (p-2)/\{1 + (\lceil m/2 \rceil - 1)(p-1)\}$

$< 1/(\lceil m/2 \rceil - 1)$

For m=200, $s_{avg} < 1/99$.

The number of disk accesses in an insertion is $h+2s+1$, where s is the number of nodes split.

So the average disk accesses is

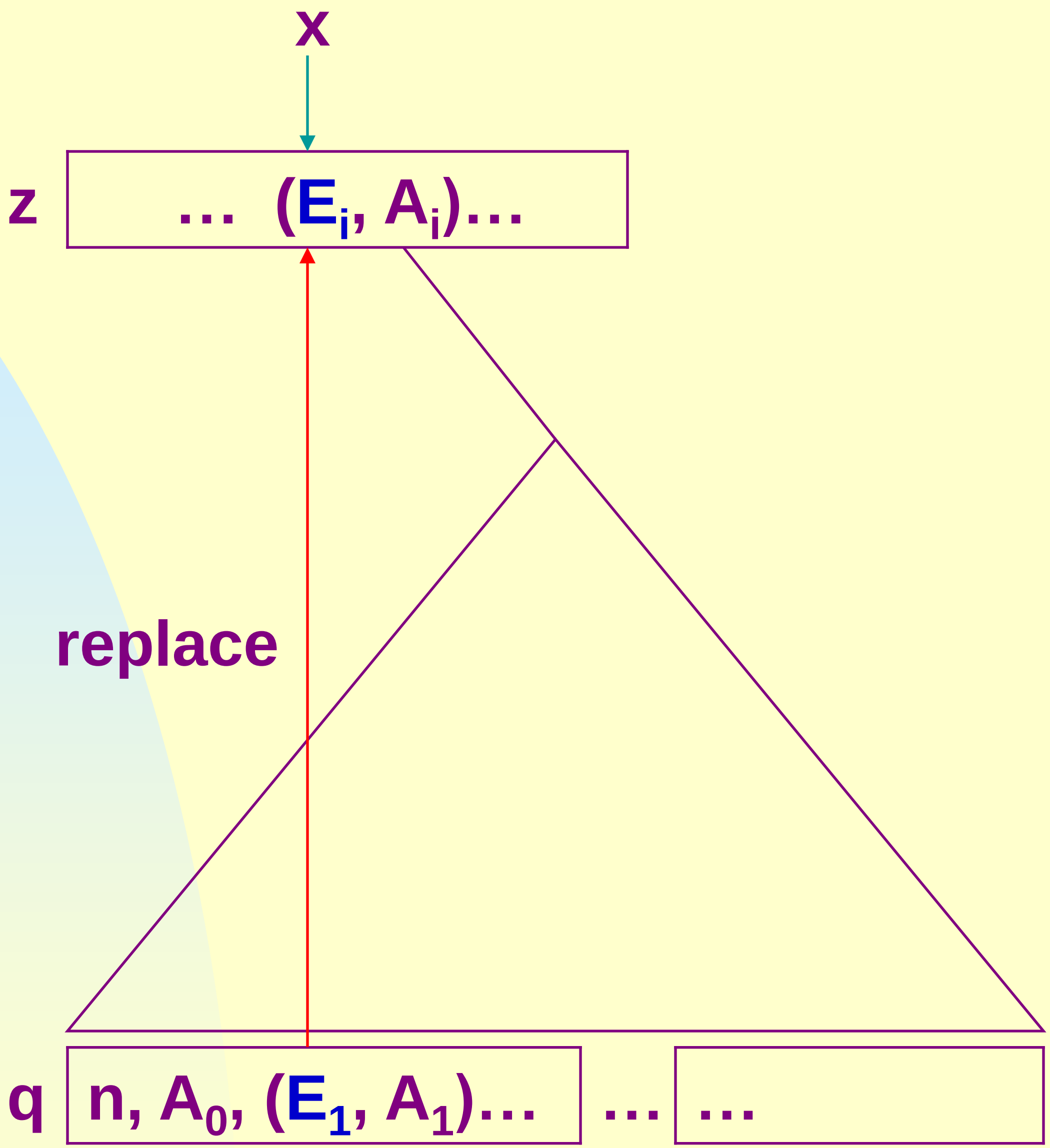
$$h+2\,s_{avg}+1 < h+2/99+1 \approx h+1.$$

11.2.4 Deletion from a B-Tree

Suppose the key of the element to be deleted is x.

First, search for x. If x is in a nonleaf node z and x = E_i.K, then the corresponding element may be replaced by either the element with smallest key in the subtree A_i or the element with largest key in the subtree A_{i-1}. Both are in leaf nodes.

The deletion from a nonleaf node is thus transformed into a deletion from a leaf, as shown in the next slide.


x
z
… (E_i, A_i)…
replace
q
n, A_0, (E_1, A_1)…
…
…

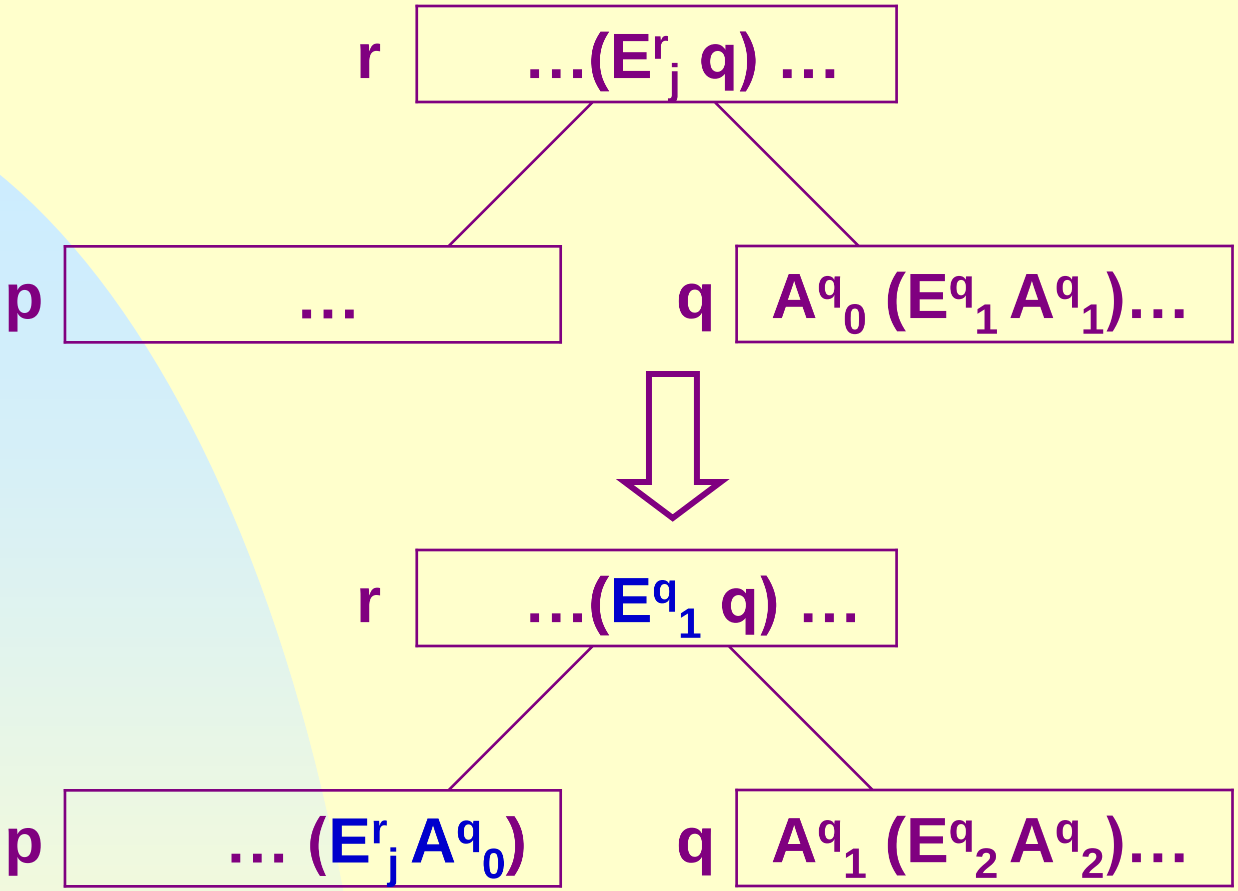
4 cases of deleting from a leaf node p:

(1) p is the root, if p is left with at least 1 key, p is written to disk, done. Otherwise the B-tree is empty following the deletion.

In the remaining cases, p is not the root.

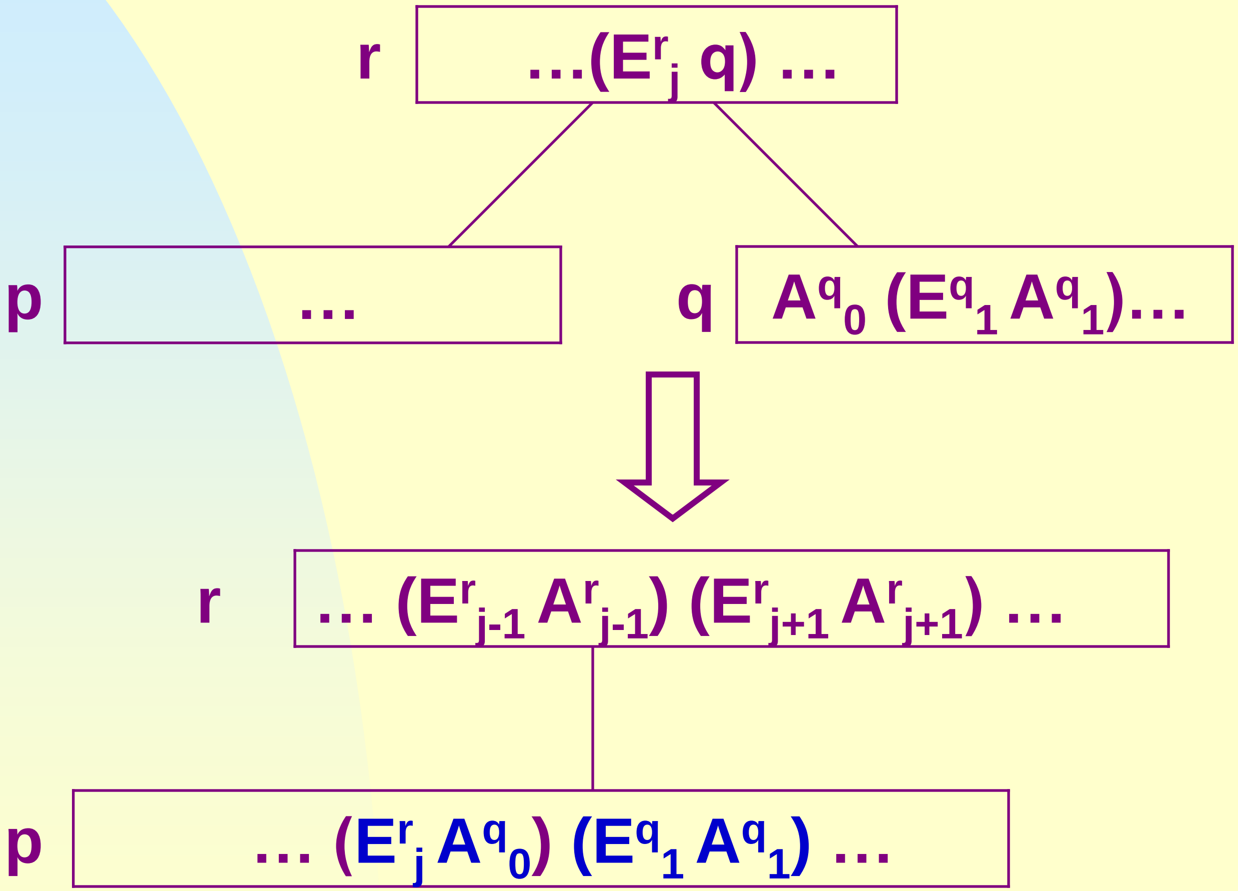
(2) Following the deletion, p has at least $\lceil m/2 \rceil$-1 keys. The modified p is written to disk, done.

(3) p has $\lceil m/2 \rceil$-2 keys, and its nearest sibling, q, has at least $\lceil m/2 \rceil$ keys. A rotation is performed as shown in the next slide (suppose q be the nearest right sibling of p, r be parent of p and q).



the changed p, q and r are written to disk, done.

(4) p has $\lceil m/2 \rceil$-2 keys, and q has $\lceil m/2 \rceil$-1 keys. A combination is performed as shown below:



Now p has ($\lceil m/2 \rceil$-2)+ ($\lceil m/2 \rceil$-1)+1=2$\lceil m/2 \rceil$-2 $\leq$ m-1 keys. p is written to disk.

The number of keys in the parent, r, has been reduced by 1.

If r does not become deficient (i.e., it has at least 1 element if it is the root and $\lceil m/2 \rceil$-1 elements if it is not the root), the changed r is written to disk, done.

When r becomes deficient, if it is the root, it is discarded. Otherwise r has $\lceil m/2 \rceil$-2 keys, we can first attempt a rotation with one of its

nearest siblings. If this is not possible, a combine is done. This process of combining can continue up the B-tree until the children of the root are combined.

A high-level description of the deletion algorithm is given in the next slide:

// Delete element with key x from a B-tree of order m

Search the B-tree for node p containing the element with key x**;**

if (there is no such node p) **return;** // no element to delete

Let p be of the form: n, A_0, (E_1, A_1), …, (E_n, A_n) and E_i.K=x**;**

if (p is not a leaf) **{**

Replace E_i with the smallest key element in subtree A_i**;**

write the altered p to disk**;**

Let p be the leaf of A_i from which the smallest was taken**;**

Let p be of the form: n, A_0, (E_1, A_1), …, (E_n, A_n)**;**

i = 1**;**

}

// the following deletes E_i from leaf node p

```
Delete (E_i, A_i) from p; n--;
while (n<⌈m/2⌉-1) && (p!=root)
    if (p has a nearest right sibling q) {
      Let q: n_q, A^q_0, (E^q_1 A^q_1),..., (E^q_nq A^q_nq);
      Let r: n_r, A^r_0, (E^r_1 A^r_1),..., (E^r_nr A^r_nr) be parent of p and q;
      Let A^r_j =q and A^r_j-1=p;
      if (n_q >= ⌈m/2⌉) { // rotation
        (E_n+1, A_n+1)= (E^r_j , A^q_0); n=n+1;   // update p
        E^r_j = E^q_1;                           // update r
        (n_q,A^q_0, (E^q_1 A^q_1),...)=(n_q-1, A^q_1, (E^q_2 A^q_2),...); //update q
        write p, q, r to disk; return;
      }
      // combine nodes p, E^r_j, and q
      s=2*⌈m/2⌉-2;
```

Delete (E_i, A_i) from p**;** n--**;**

while (n<$\lceil m/2 \rceil$-1) **&&** (p!=root)

if (p has a nearest right sibling q) **{**

Let q: n_q, A^q_0, $(E^q_1\ A^q_1), \ldots, (E^q_{nq}\ A^q_{nq})$**;**

Let r: n_r, A^r_0, $(E^r_1\ A^r_1), \ldots, (E^r_{nr}\ A^r_{nr})$ be parent of p and q**;**

Let A^r_j =q and A^r_{j-1}=p**;**

if (n_q >= $\lceil m/2 \rceil$) **{** // rotation

$(E_{n+1}, A_{n+1}) = (E^r_j, A^q_0)$**;** n=n+1**;** // update p

$E^r_j = E^q_1$**;** // update r

$(n_q, A^q_0, (E^q_1\ A^q_1), \ldots) = (n_q-1, A^q_1, (E^q_2\ A^q_2), \ldots)$**;** //update q

write p, q, r to disk**; return;**

}

// combine nodes p, E^r_j, and q

s=2*$\lceil m/2 \rceil$-2**;**

write s, A_0, (E_1 A_1),…,(E_n A_n), (E^r_j A^q_0), (E^q_1 A^q_1), …,(E^q_{nq} A^q_{nq}) to disk as node p**;**

// update for next iteration

free(q)**;**

(n, A_0,…)=(n_r-1, A^r_0,…,(E^r_{j-1} A^r_{j-1}), (E^r_{j+1} A^r_{j+1})…)**;**

p=r**;**

} // end of **if** p has a nearest right sibling

else { // p must have a left sibling, this is symmetric to the

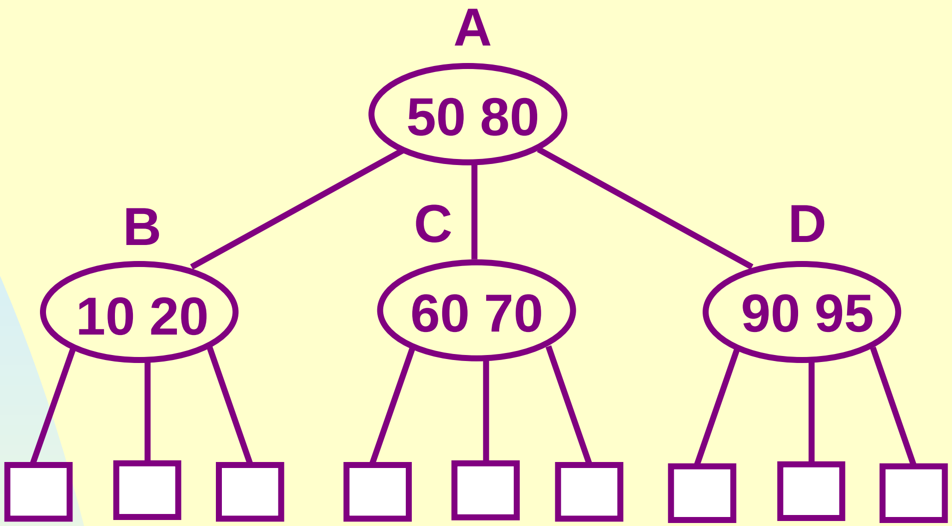
// case where p has a right sibling

} // end of **if-else** and **while**

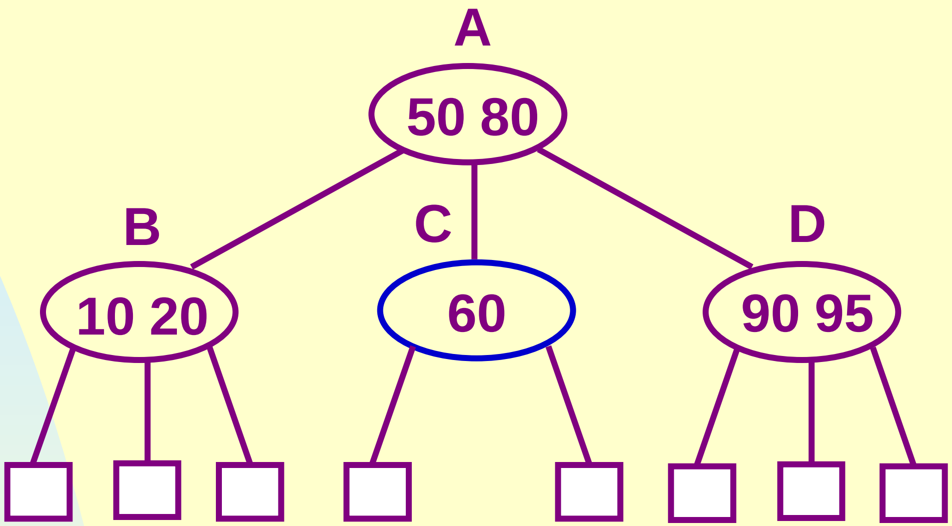
if (n) **write** p: (n, A_0, (E_1, A_1), …, (E_n, A_n))**;**

else { root= A_0**; free**(p)**;}** // change root, always combine to left

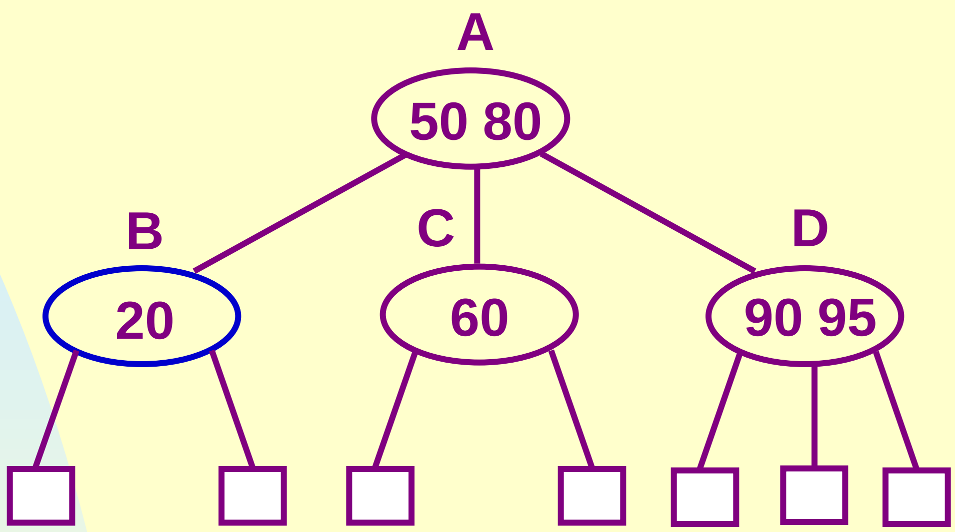
Example 11.2:



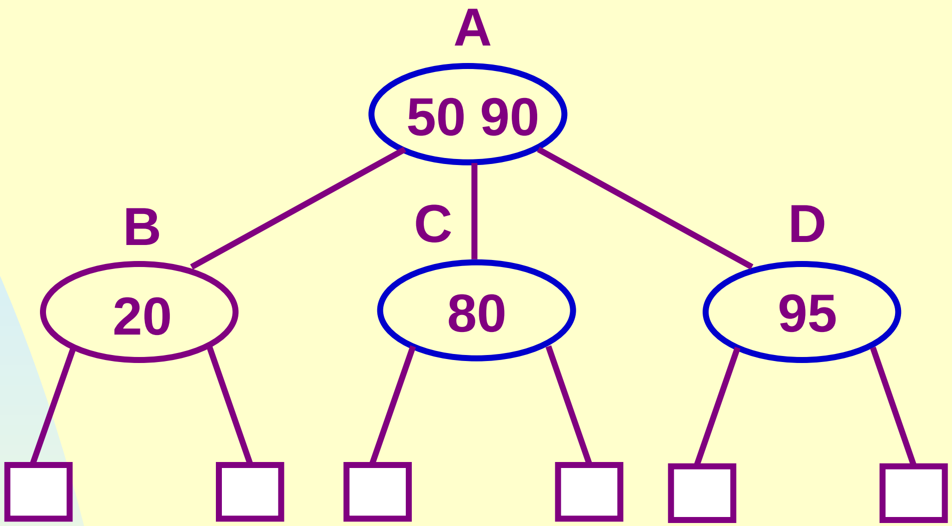
An initial 2-3 tree



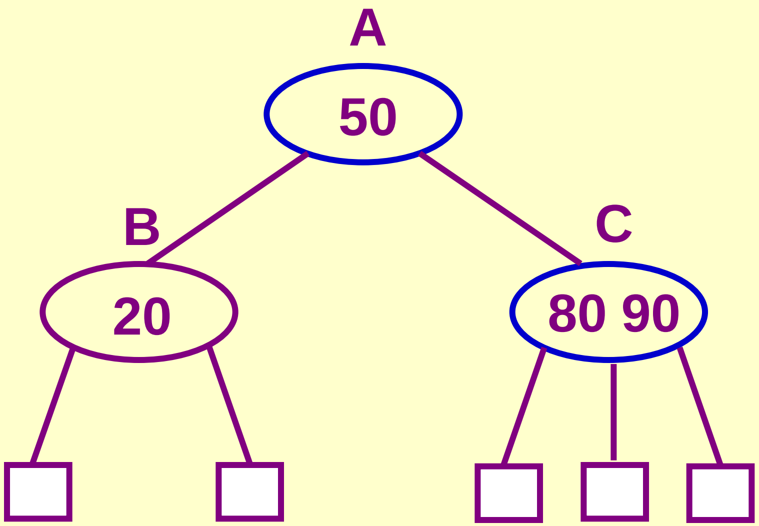
70 deleted



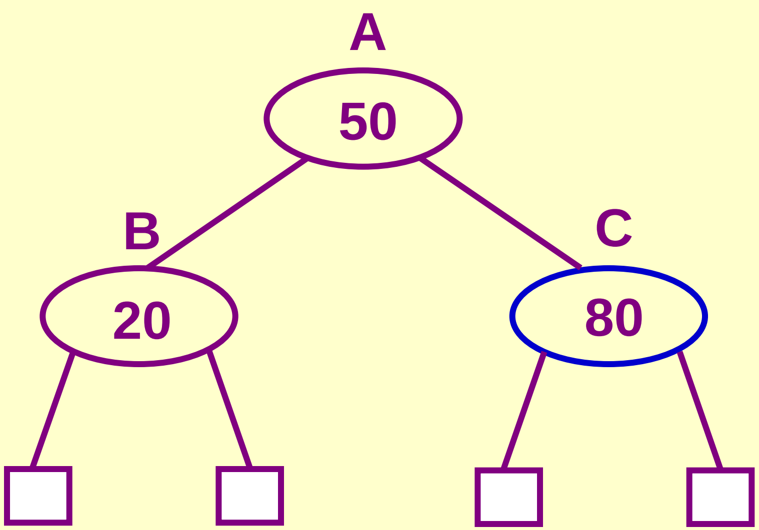
10 deleted



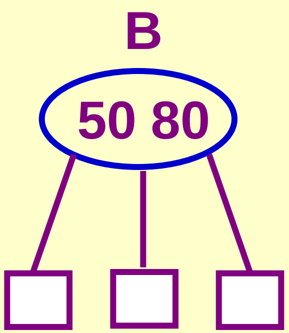
60 deleted



95 deleted



90 deleted



20 deleted

Analysis of B-tree Deletion:

h+1 disk accesses for finding the node from which the key is to be deleted and transforming the deletion to a one from a leaf.

In the worst, a combine takes place at each of the last h-2 nodes on the root-to-leaf path, and a rotation takes place at the 2^{nd} node on this path. The combines need: h-2 disk accesses for sibling and h-2 for writing out. The rotation needs 1 for sibling and 3 for writing out.

Total number of disk accesses is 3h+1.

Exercises: P623-2, 4

6.1.3 Graph Representations

6.2.1—2 Elementary Graph Operations

6.4.3 All-Pairs Shortest Paths

7.3 Quick Sort

7.5.1-2 Iterative Merge Sort

7.10.2 k-way Merging

The end of the course

Thank you for your cooperation!