EE 441 Data Structures

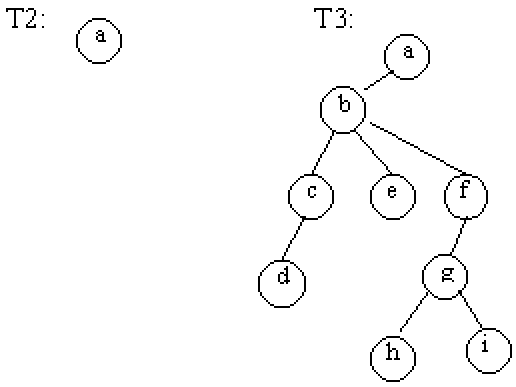
Lecture 8: Trees

Trees

- Linear lists, structures:
  - Arrays, stacks, queues, linked lists
  - Unique first and last element, each element has a unique successor
- Non-linear structures:
  - A member might have multiple successors
  - Trees:
    - Nodes and branches
    - Flow from root to leaves (outer nodes)

Trees

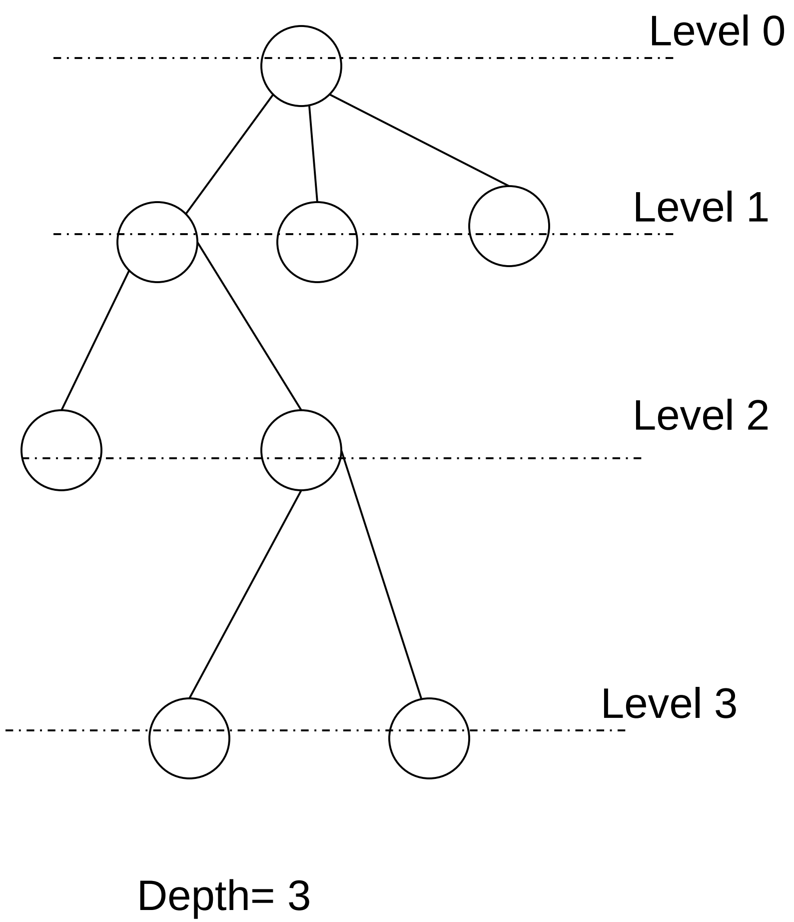
- A tree is a set of nodes:
- It can be a null tree without any nodes
- one node designated as the root and the remaining nodes partitioned into smaller trees, called sub-trees.



- Example:
  - T1={} (NULL Tree)
  - T2={a} a is the root, the rest is T1
  - T3={a, {b,{c,{d}},{e},{f,{g,{h},{i}}}}

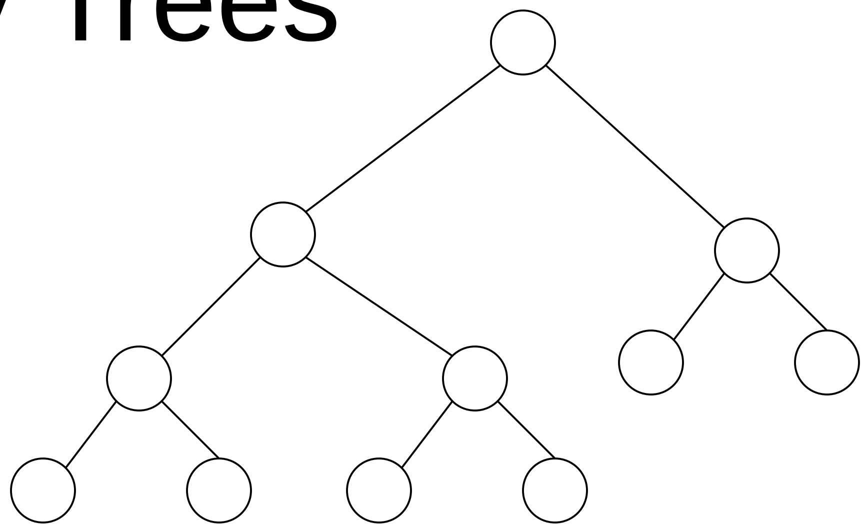
Trees

- The level of a node is the length of the path from the root to that node
- The depth of a tree is the maximum level of any node in the tree
- The degree of a node is the number of partitions in the subtree which has that node as the root
- Nodes with degree=0 are called leaves



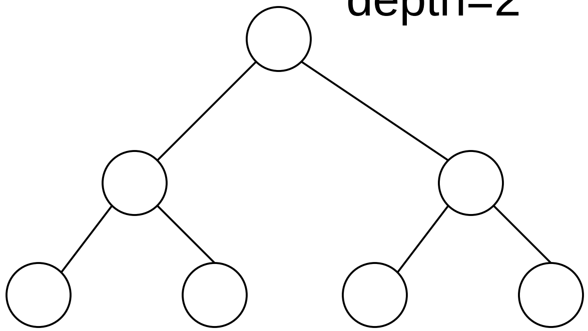
Binary Trees

- A tree in which the maximum degree of any node is 2.
- Uniform structure: Efficient scan and search
- A binary tree may contain up to 2^n nodes at level n.



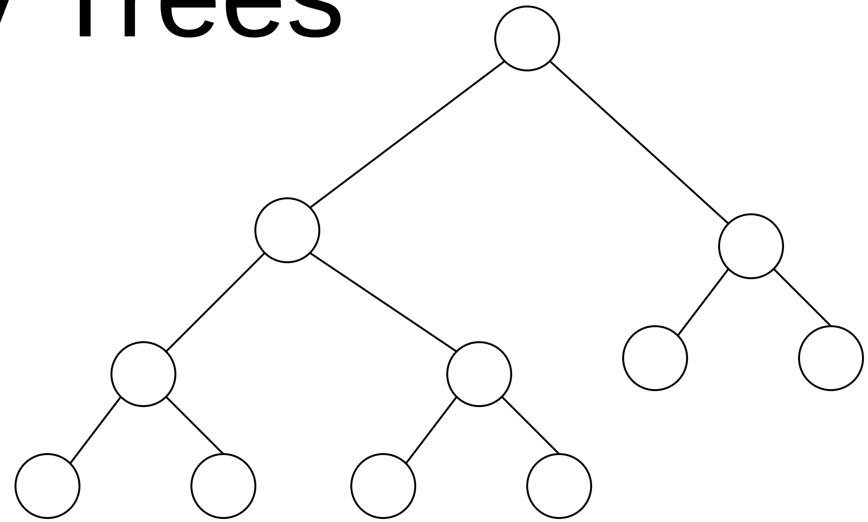
depth=3

depth=2

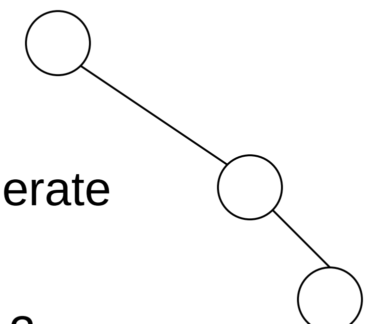


Binary Trees

- A complete binary tree of depth N has 2^k nodes at levels k=0,…,N-1 and all leaf nodes at level N occupy leftmost positions.
- If level N has 2^N nodes as well, then the complete binary tree is a full tree.
- If all nodes have degree=1, the tree is a degenerate tree (or simply linked list)



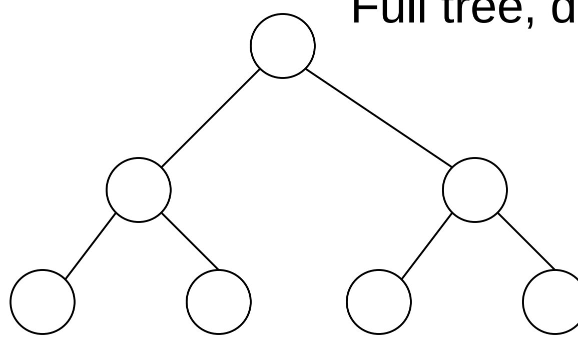
Complete tree, depth=3



Degenerate tree,

depth=2

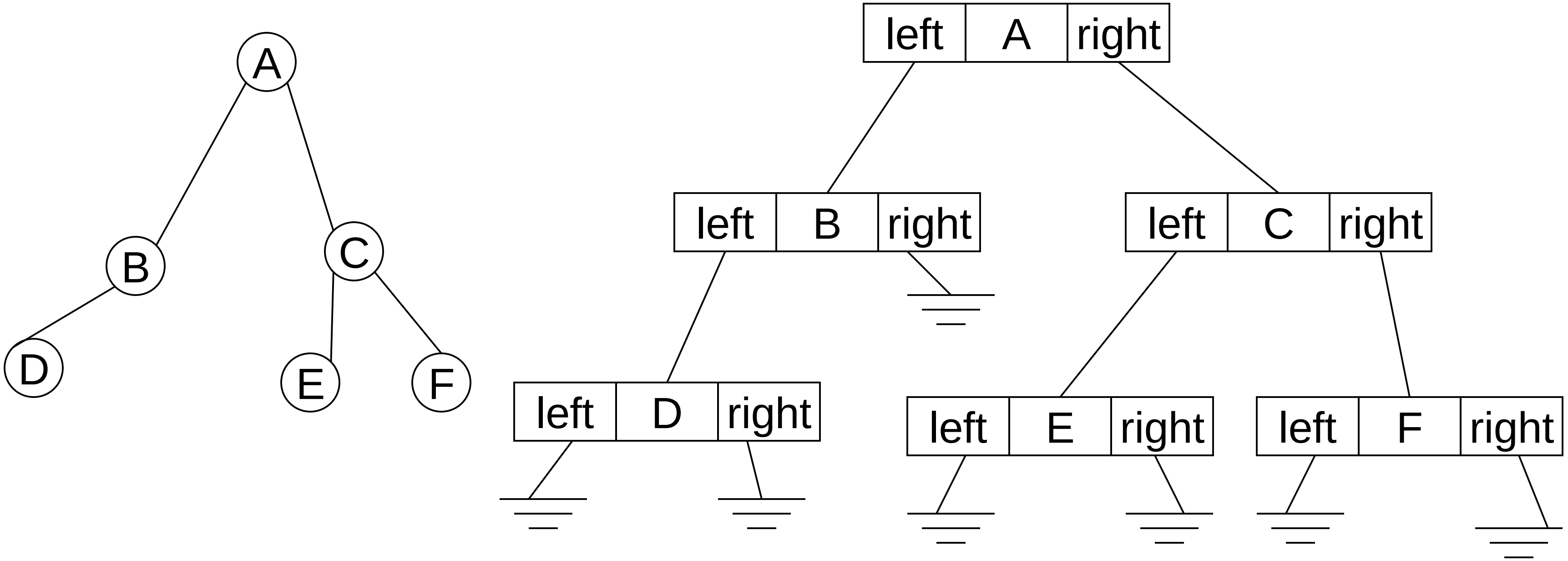
Full tree, depth=2



Examples: Binary Trees

- a degenerate tree of depth 5 has 6 nodes
- a full tree of depth 3 has 1+2+4+8=15 nodes
- a full tree of depth N has 2^{N+1}-1 nodes
- a complete tree of depth N has $2^N \le m \le 2^{N+1}-1 < 2^{N+1}$ nodes
- Depth of a complete tree with m nodes: $k \le \log_2 m \le k+1$
- The maximum depth in a tree with 5 nodes is 4: degenerate
- The minimum depth in a tree with 5 nodes is 2: complete
- $\mathrm{int}(\log_2 5)=\mathrm{int}(2.32)=2$

Binary Trees: Data structure



Tree Node

```
template <class T>
class TreeNode
{private:
TreeNode<T> *left;
TreeNode<T> *right;
public:
T data;
//constructor
TreeNode(const T &item, TreeNode<T>
   *lptr=NULL, TreeNode<T>
*rptr=NULL);
//access methods for the pointer
   fields
TreeNode<T>* Left(void) const;
TreeNode<T>* Right(void) const;
};
```

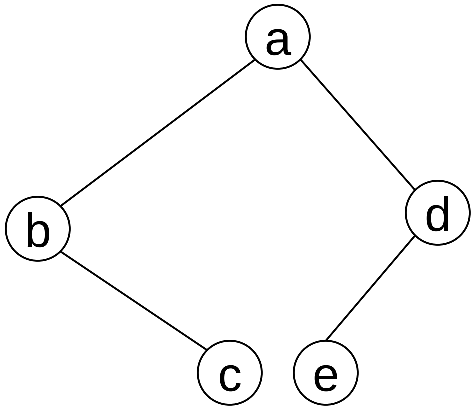
```
//constructor
template <class T>
TreeNode<T>:: TreeNode(const T
   &item, TreeNode<T> *lptr,
TreeNode<T> *rptr): data(item),
   left(lptr), right(rptr)
```

Tree Node

```
//GLOBAL function to dynamically allocate memory for a new
   object
template <class T>
Treenode<T> *GetTreeNode(T item, TreeNode<T> *lptr=NULL,
   TreeNode<T> *rptr=NULL)
{
TreeNode<T> *p;
p=new TreeNode<T> (item, lptr, rptr);
if (p==NULL) // if "new" was unsuccessful
{cerr<<Memory allocation failure"<<endl;
exit(1);
}
return p;}
// a GLOBAL function to deallocate memory template <class T>
void FreeTreeNode(TreeNode <T> *p)
{delete p;}
```

Constructor Example

```
TreeNode<char> *t;
t=GetTreeNode('a', GetTreeNode('b',NULL,
   GetTreeNode('c')), GetTreeNode('d',
   GetTreeNode('e')));
```



Traversal

- Traversal: Visit/process all data stored in the data structure
- Linked List Traversal:
  – Linear Data Structure
  – Start from the head, go from node to node using NextNode()
- Tree Traversal:
  – Non-linear Data Structure

Tree Traversal

- Two main methods:
  - Both start from the root node
  - Depth First: Recursive descend to the leafs
  - Breadth First: Iterative level by level scan

Depth First

- Root with two sub-trees identified by the left and right pointers
- Performs 3 actions:
  - Visit the node (N)
  - Recursively descend to left sub-tree (L)
  - Recursively descend to right sub-tree (R)
  - The order of these actions are different for different Depth First Algorithms
- After a descent the algorithm identifies the new node and the new sub-trees
- Descent terminates when we reach an empty tree (pointer==NULL)

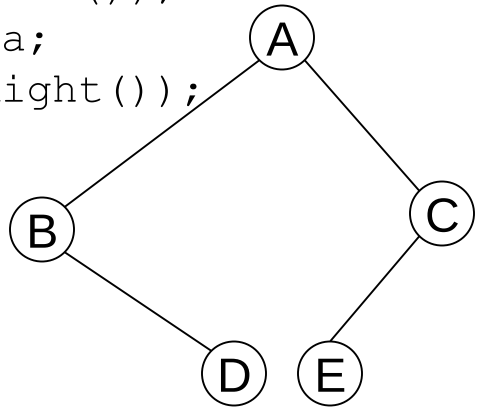
Depth First Tree Traversal

- In-order: LNR
  - Traverse left subtree
  - Visit node (i.e. process node)
  - Traverse right subtree
- Pre-order: NLR
  - Visit node (i.e. process node)
  - Traverse left subtree
  - Traverse right subtree
- Post-order: LRN
  - Traverse left subtree
  - Traverse right subtree
  - Visit node (i.e. process node)

Inorder Recursive Traversal

- Inorder: LNR
  - Traverse left subtree
  - Visit node (i.e. process node)
  - Traverse right subtree

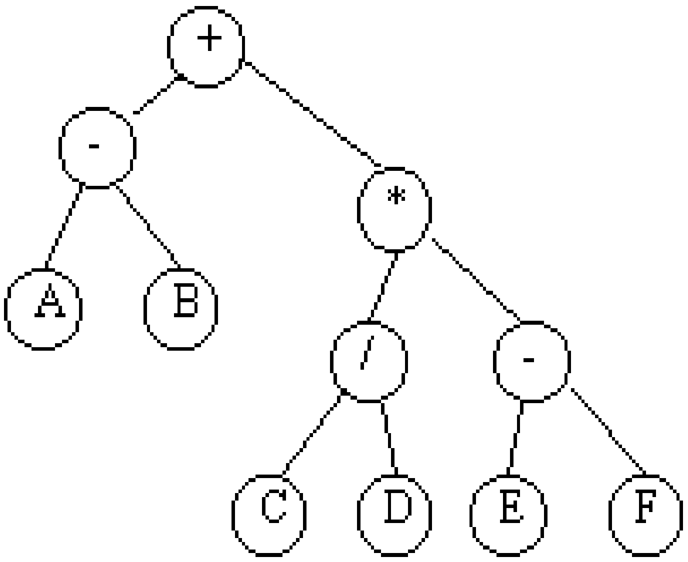
```
template<class T>
void Inorder(TreeNode<T>*t)
{
if(t!=NULL)
{
Inorder(t->Left());
cout<<t->data;
Inorder(t->Right());
}

}
```



- Descend left from A to B
- Left child of B is NULL
- Visit B
- Descend right from B to D
- D is a leaf node
- Visit D
- Visited left sub-tree of A
- Visit A
- Descend right from A to C
- Descend left from C to E
- E is a leaf node
- Visit E
- Visit C
- Done!

Depth First Tree Traversal Example

- Operands in leaves
- Operators in non-leaf nodes
- LNR:
  - (A-B)+((C/D)*(E-F))
- NLR:
  - +-AB*/CD-EF
- LRN:
  - AB-CD/EF-*+



Implementations

```
template <class T>
void InOrder(TreeNode<T>*t)
{
if(t!=NULL)
{
InOrder(t->Left())
cout<<t->data()//do what you will do
    when you visit the node
InOrder(t->Right())

}

}
```

```
template <class T>
void PreOrder(TreeNode<T>*t)
{
if(t!=NULL)
{
cout<<t->data()//do what you will do when
    you visit the node
PreOrder(t->Left())
PreOrder(t->Right())

}

}
template <class T>
void PostOrder(TreeNode<T>*t)
{
if(t!=NULL)
{
PostOrder(t->Left())
PostOrder(t->Right())
cout<<t->data()//do what you will do when
    you visit the node
}
}
```

Example

- Count the leaves of a binary tree
- Use Post-order traversal (LRN)
- Idea:
  - Keep a count variable
  - Increment at each visit when ıt ıs a leaf node
- Problem:
  - Each visit is a separate recursive call.
  - How to pass and modify the count variable between functions?

Example

```
template <class T>
void CountLeaf(TreeNode<T> *t, int& count)
{
if (t!=NULL)
{//using postorder traversal
CountLeaf(t->Left(),count);
CountLeaf(t->Right(),count);
//visiting a node means incrementing if leaf
if (t->Left()==NULL&&t->Right()==NULL)
count++;
}
}
```

PASS BY REFERENCE:
To pass and modify information among recursive calls

Example

- Find the depth of a binary tree
- Idea:
  - Recursively descend to the leaves (depth =0 at the leaves)
  - As you come back up to root increment depth
  - Pass information with a return value from function

Example

```
template <class T>
int Depth(TreeNode<T> *t)
{int depthleft, depthRight,
   depthval;
if (t==NULL)
depthval=-1; // if empty, depth=-1
else
{depthLeft=Depth(t->left());
depthRight=Depth(t->Right());
depthval=1+(depthleft>depthRight?dep
   thLeft:depthRight));
}
return depthval;
}
```

```
if(depthleft>depthRight)
depthLeft
else
depthRight

CONDITION? True-case-EXP:False-case-
   EXP
```

Examples

- Find the level of the leaf node at the minimum level

```
template <class T>
int minLeafdepth(TreeNode<t>* t)
{}
```

- Find the minimum key value stored in a binary tree pointed by t

```
template <class T>
int minval(TreeNode<t>* t)
{}
```

Breadth First

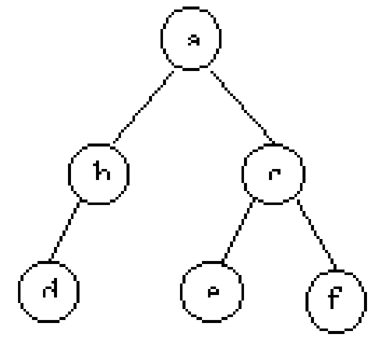
- Scan level by level: visit all nodes at the same level, then descend next level
- Iterative algorithm:
- A queue to hold the items
- Algorithm Level-Traverse

1. Insert root node in queue
2. While queue is not empty

2.1. Remove front node from queue and visit it

2.2. Insert Left child

2.3. Insert right child



Traversal sequence
a,b,c,d,e,f

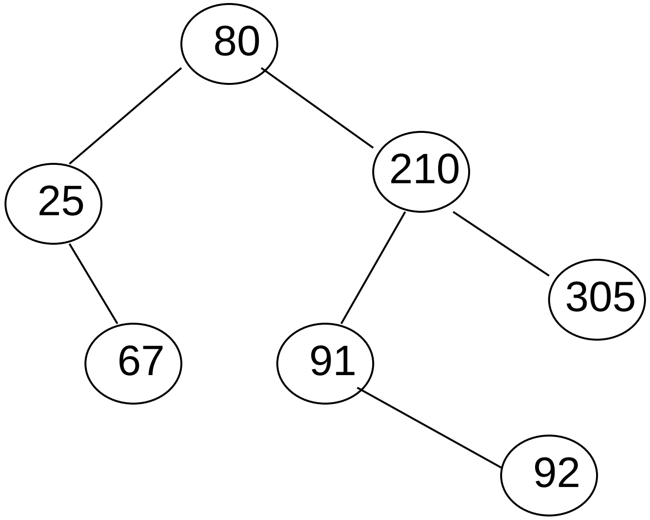
Breadth First

```
// traverse the list by level by level and visit each node
template <class T>
void LevelScan(TreeNode<T> *t, void visit(T& item))
{
   // store siblings of each node in a queue so that they are
   // visited in order at the next level of the tree
   Queue<TreeNode<T> *> Q;
   TreeNode<T> *p;//temporary variable
   // initialize the queue by inserting the root in the queue
   Q.QInsert(t);
   // continue the iterative process until the queue is empty
   while(!Q.QEmpty())
   {
      // delete front node from queue and execute visit function
      p = Q.QDelete();
   cout<<p->data()//do what you will do when you visit the node
      // if a left child exists, insert it in the queue
      if(p->Left() != NULL)
       Q.QInsert(p->Left());
      // if a right child exists, insert next to its sibling
      if(p->Right() != NULL)
       Q.QInsert(p->Right());
   }
}
```

Binary Search Tree

- A BST is a BT in which data values in the left subtree of every node are "less than" the data value in the node and those in the right subtree are "greater".

  e.g.

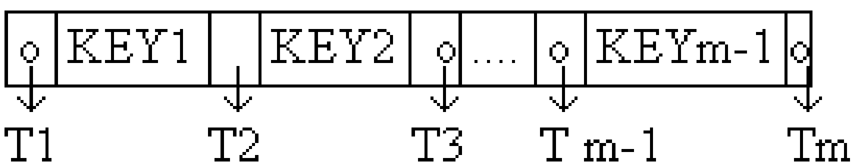


NOTE: Inorder traversal (LNR) generates ascending sequence

A BST is a two-way search tree

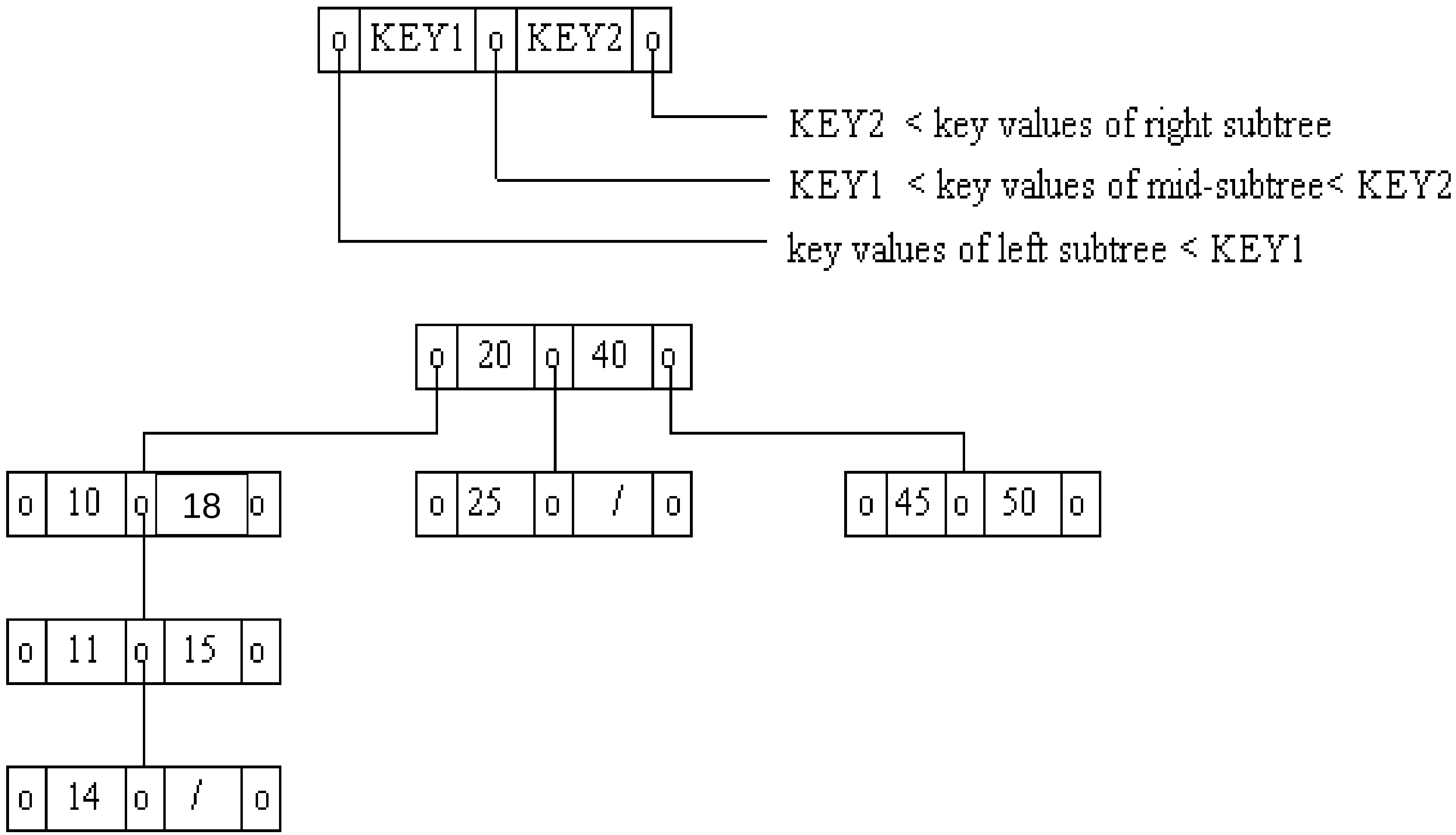
M-way Search Tree

- Definition: An m_way search tree is a tree in which all nodes are of degree<=m. (It may be empty). A non empty m_way search tree has the following properties:
- a) It has nodes of type:



- b) key1 < key2 <...< key(m-1)
- in other words, $key_i<key_{(i+1)}$, $1<=i<m-1$
- c) All Key values in subtree Ti are greater than Key_{i-1} and less than Key_i

3-way Search Tree

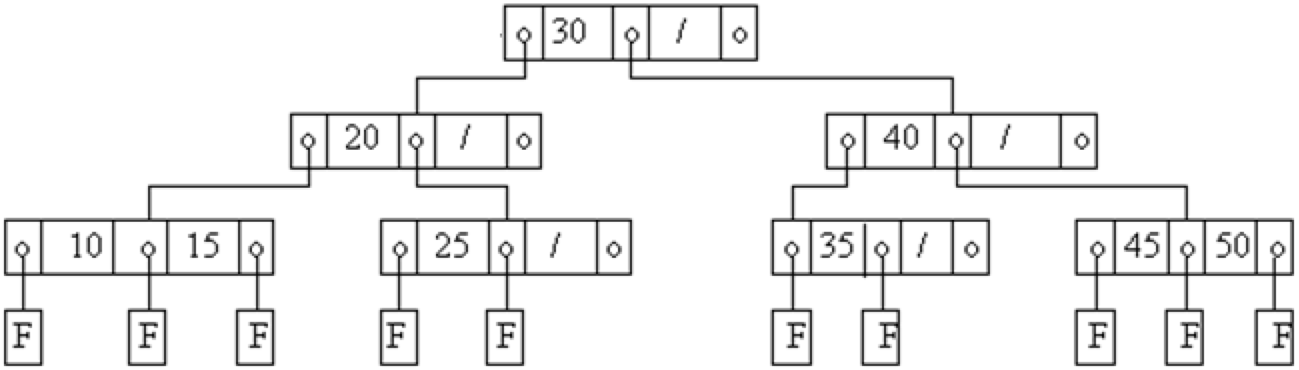


M-way Search Tree

- Failure nodes are the nodes where an unsuccessful search terminates

- For all keys stored:
  - There are non-empty subtrees at the right and left of the key

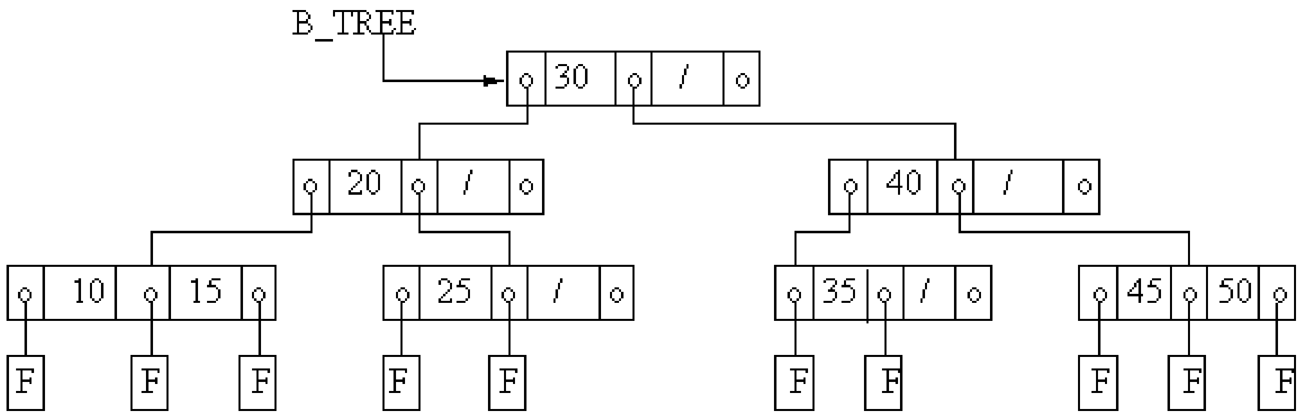
  OR
  - A failure node in place of an empty subtree



(Balanced) B-Trees

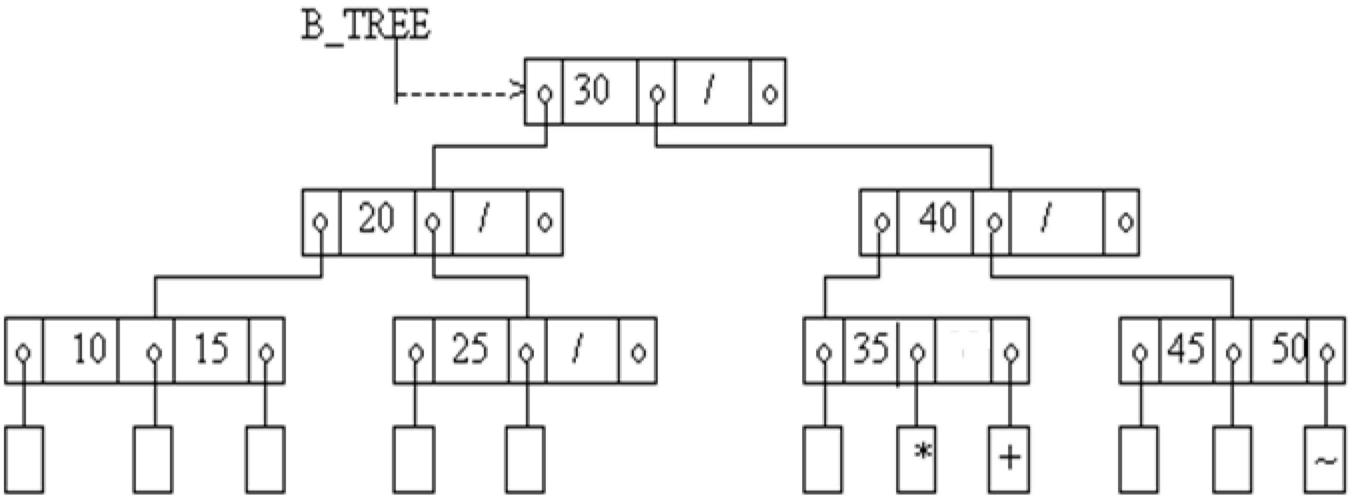
A B-tree of order m is an m-way search tree (possibly empty) satisfying the following properties (if it is not empty)

- All nodes other than the root node and leaf nodes have at least, $\left\lceil \frac{m}{2} \right\rceil$ children
- The tree is balanced➔ Any failure in any search must always end at the same level
  - All the leaf nodes are at the same level
  - All (failure) nodes are at the same level



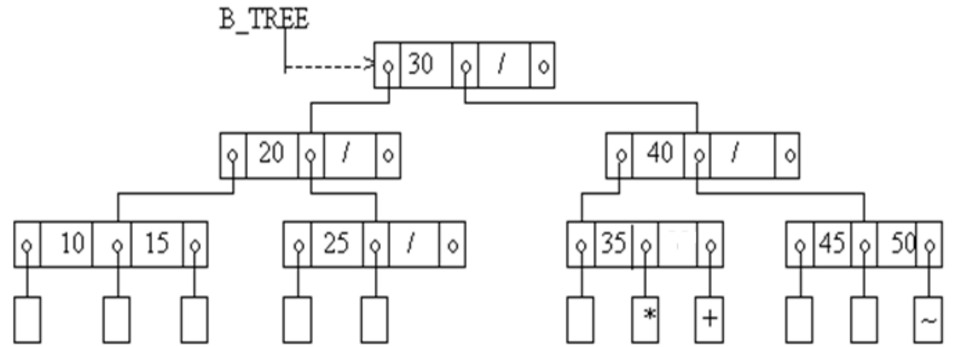
Inserting Nodes

- We want to insert a new key value into a B-tree:
  - The resulting tree must also be a B-tree. (It must be balanced.)
  - We'll always insert at the leaf nodes.
- Example1: Insert 38 to the B_tree

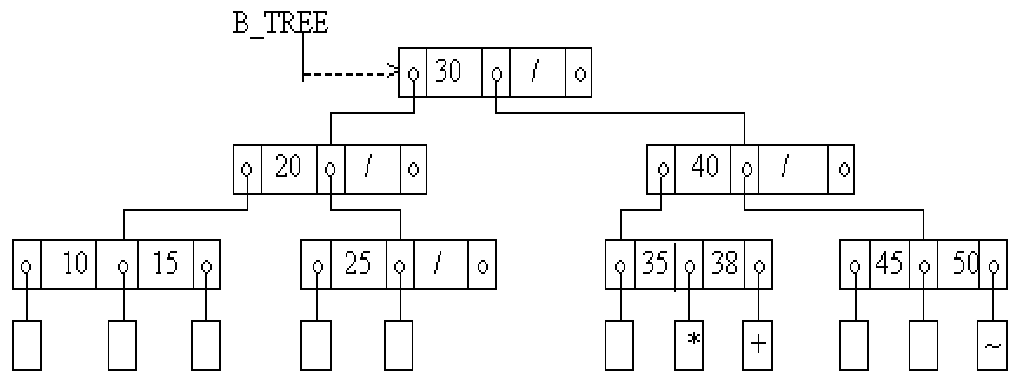


Inserting Nodes: Example 1

- Insert 38 to the B_tree

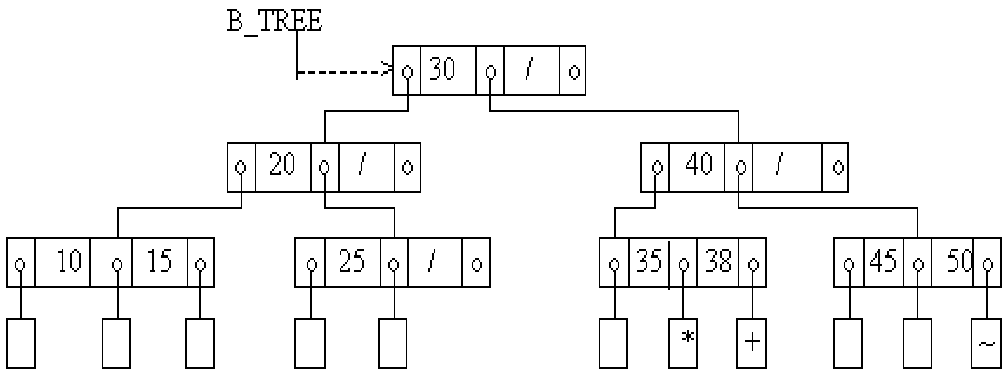


1. Search for 38 in the given b_tree. It is an unsuccessful search.
2. We hit the failure node marked with "*" .
3. The parent of that failure node has only one key value so it has space for another one.

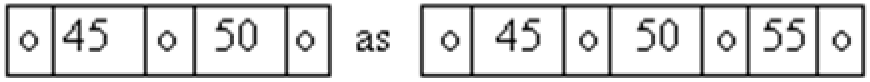


4. Insert 38 there , add a new failure node, which is marked as "+" in the following figure, and return.

Inserting Nodes: Example 2

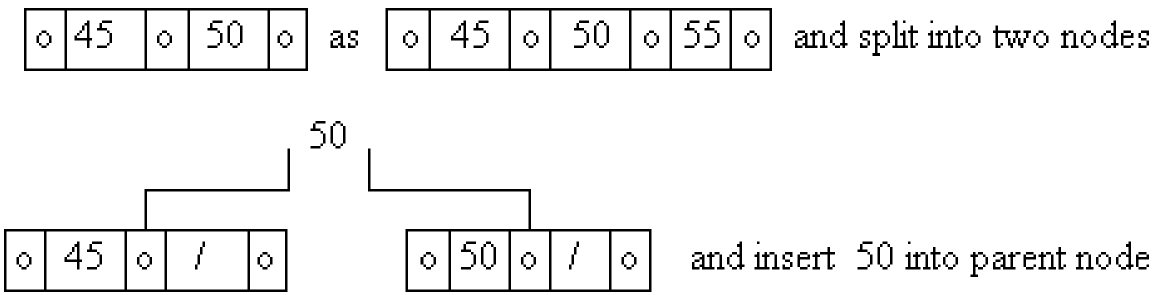


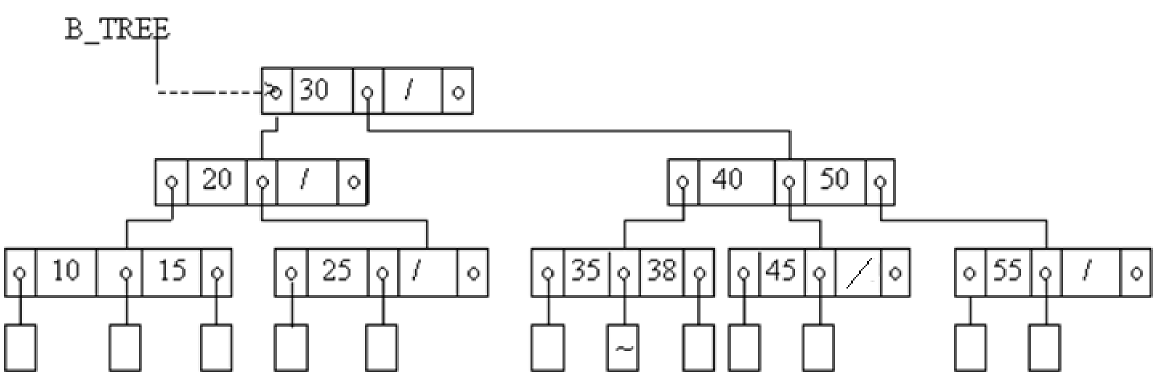
- Example2: Now, insert 55 to this B_tree.
- We do the search and hit the failure node "~".
- However, it's parent node does not have any space for a key value.
- We create temporarily a new node with m key values and insert 55 in this new node



Inserting Nodes: Example 2

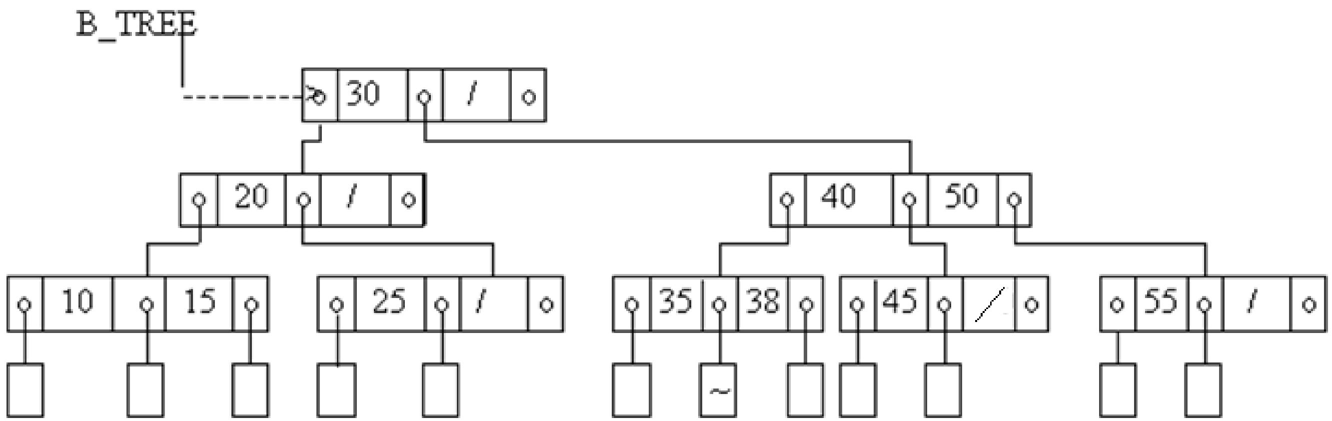
- We then split this temporary node and move the center element out and insert it into its parent node





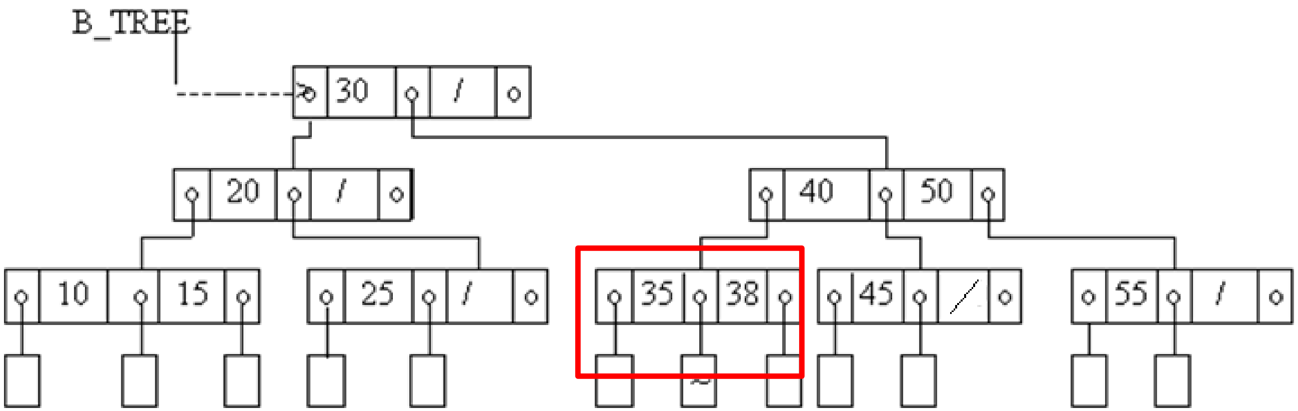
Final B-Tree

Inserting Nodes: Example 3

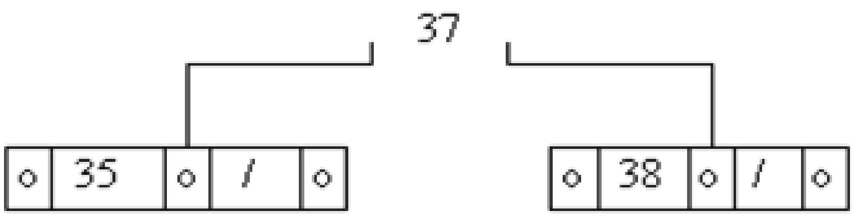


- Insert 37
- We search for 37, and hit a failure node between 35 and 38.

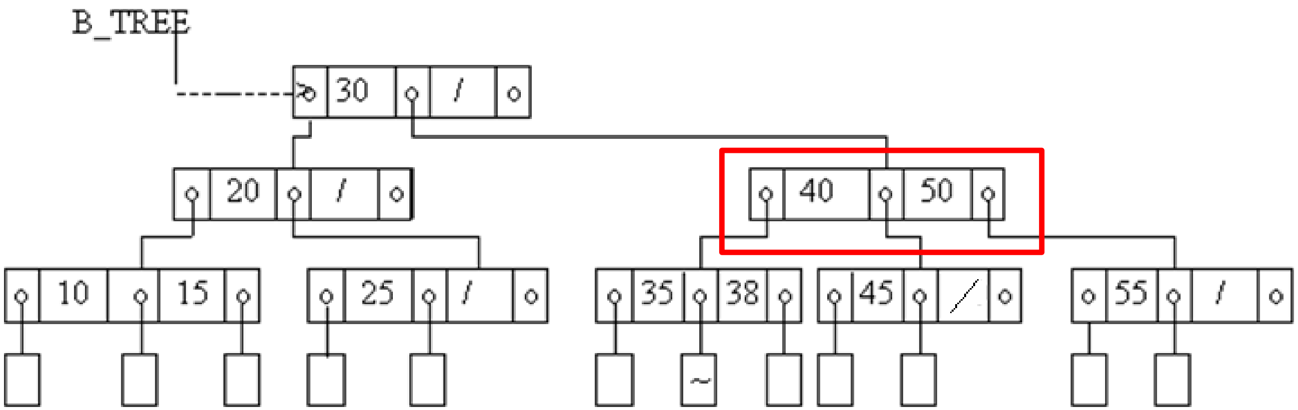
Inserting Nodes: Example 3



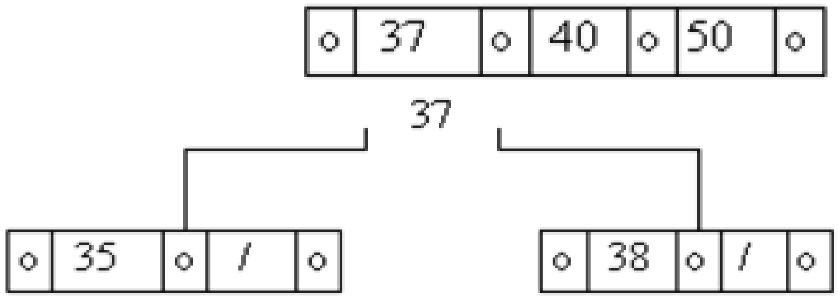
split it,



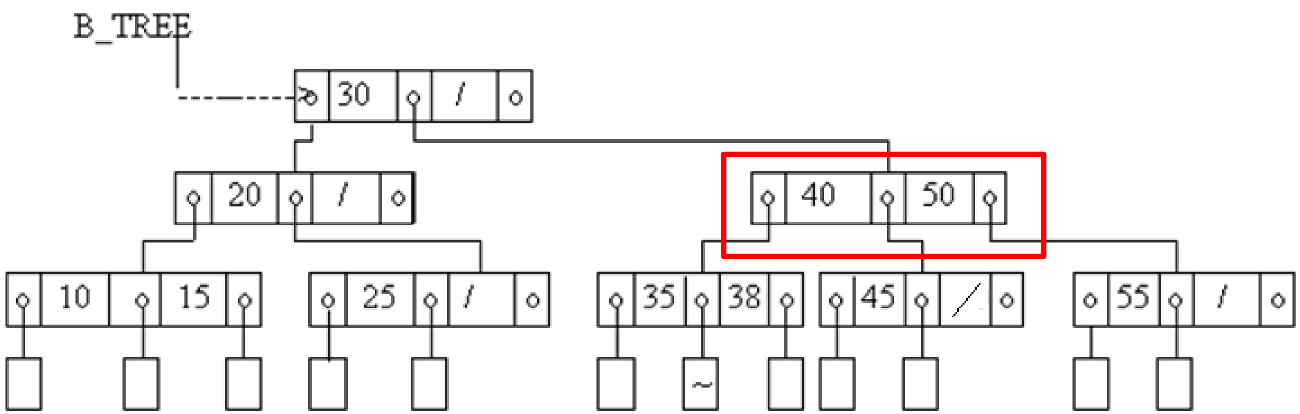
Inserting Nodes: Example 3



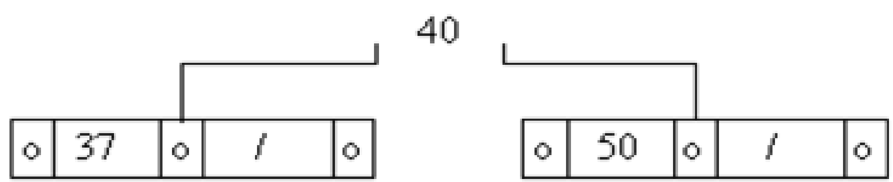
insert 37 to its parent



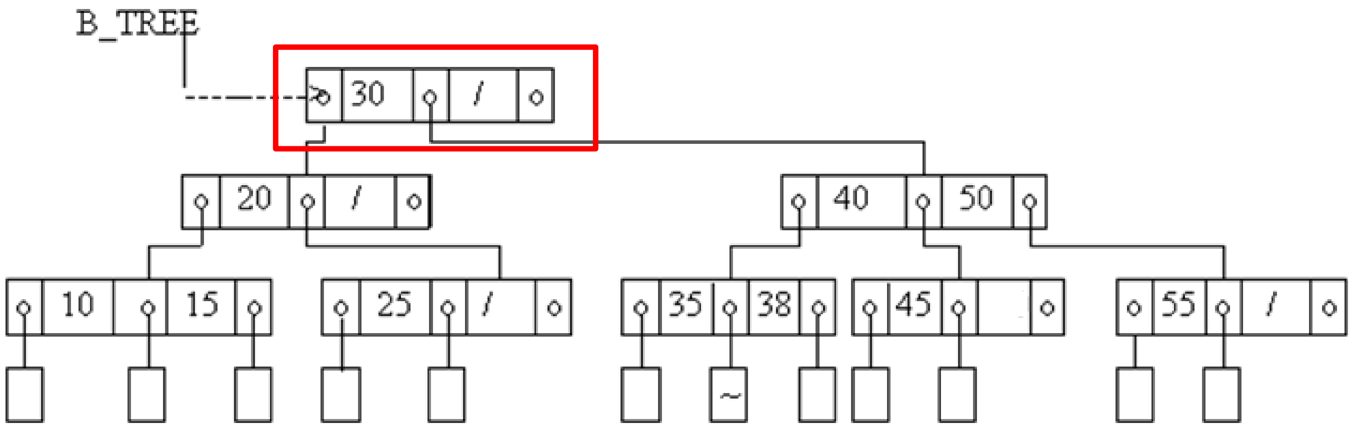
Inserting Nodes: Example 3



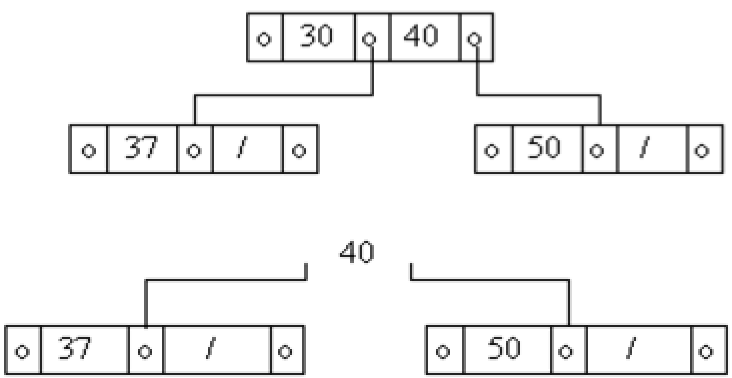
since there is no space for a new key , split it again



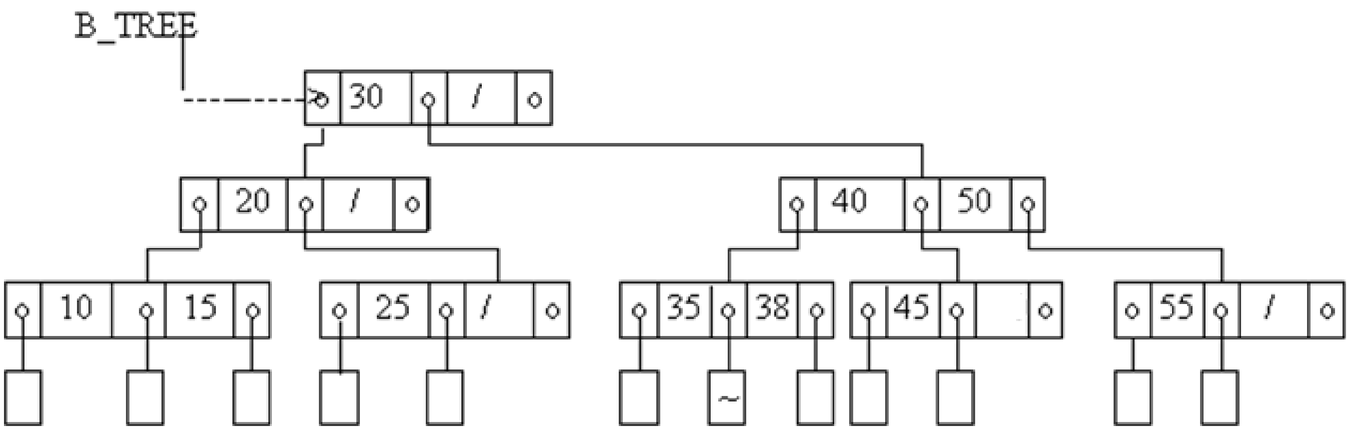
Inserting Nodes: Example 3



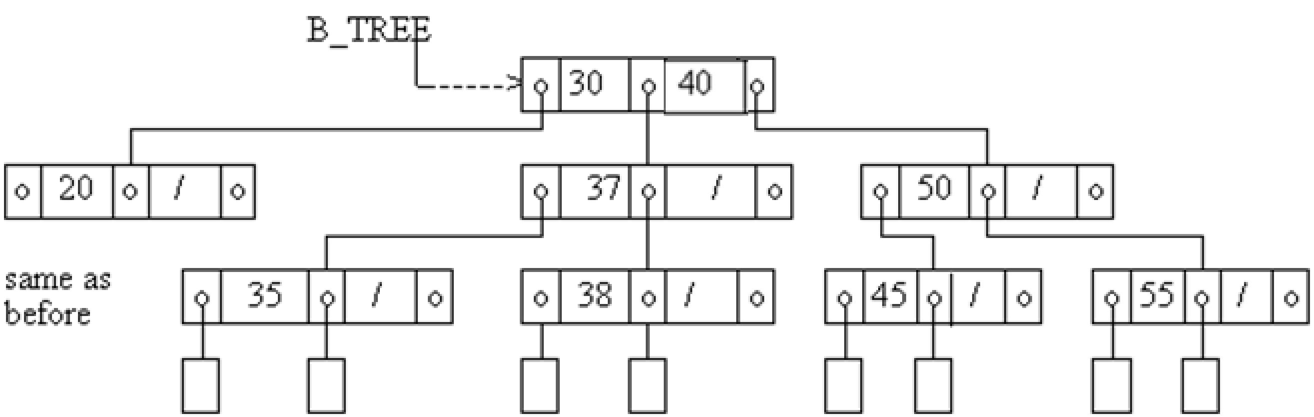
And this time insert 40 into its parent



Inserting Nodes: Example 3



Before Insertion:
Depth=2



After Insertion :
Depth=2

B-Trees

- The depth of the B-Tree increases very slowly
- Search time increases with depth in m-way search trees ➔ Fast search time
- When does the depth of a B-Tree increment?
- When we split the root node
- Example: Insert 34, 32, and 33

EE 441 Data Structures

Lecture 8: Trees