EE 441 – CH7

TREES

Instructor: UĞUR HALICI

TREES

A tree is a set of nodes which is either null or with one node designated as the root and the remaining nodes partitioned into smaller trees, called sub-trees.

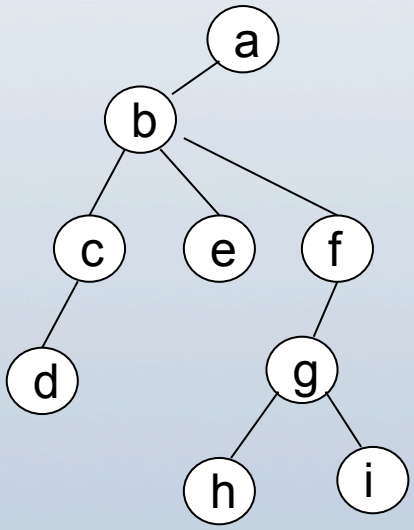
Example:

T1={} (NULL Tree)

T2={a} a is the root, the rest is T1

T3={a, {b,{c,{d}},{e},{f,{g,{h},{i}}}}

graphical representation of T3:



- 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 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

Example:

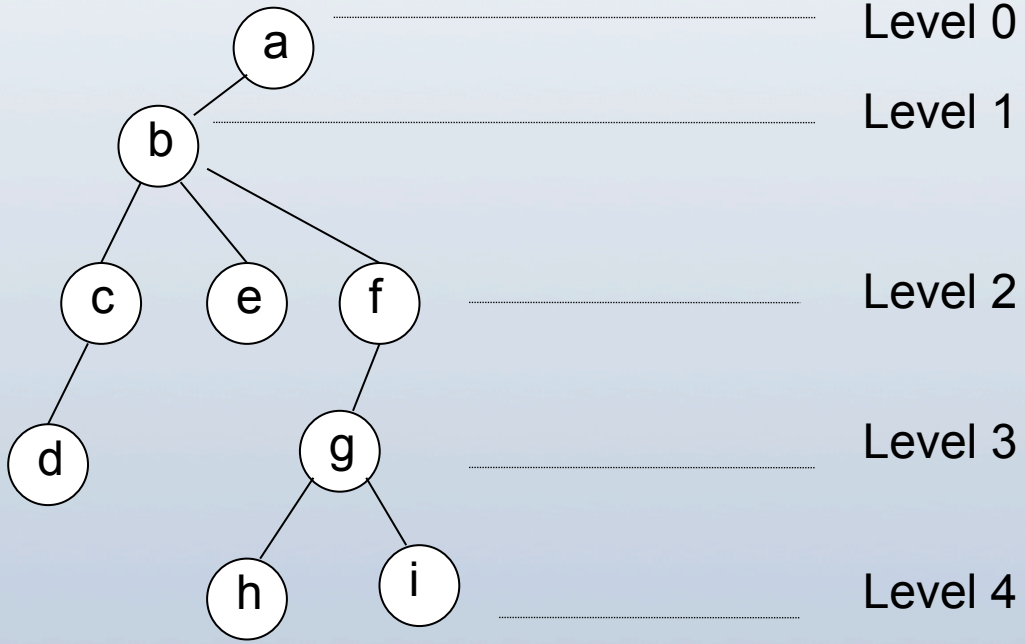
depth=4

degree(a)=1

degree(b)=3

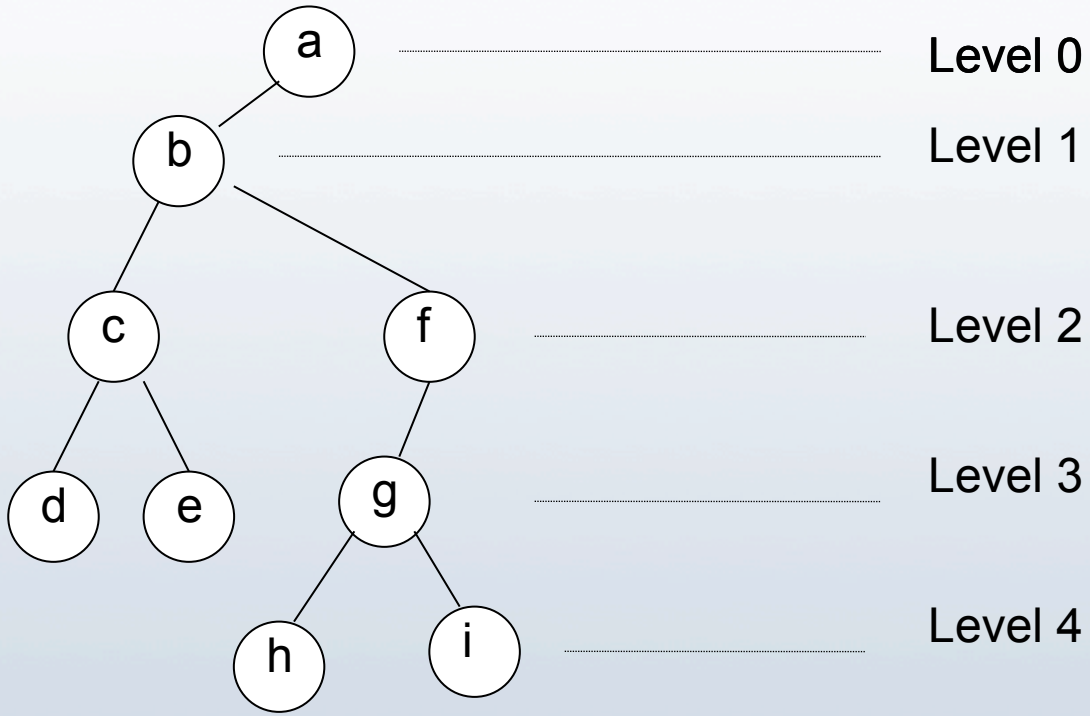
degree(e)=0

leaves: e,d,h,i



BINARY TREE

Binary tree is a tree in which the maximum degree of any node is 2.



A binary tree may contain up to 2^n nodes at level n.

A complete binary tree of depth N has 2k nodes at levels k=0,…,N-1 and all leaf nodes at level N occupy leftmost positions.

If level N has 2N 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)

e.g. a degenerate tree of depth 5 has 6 nodes

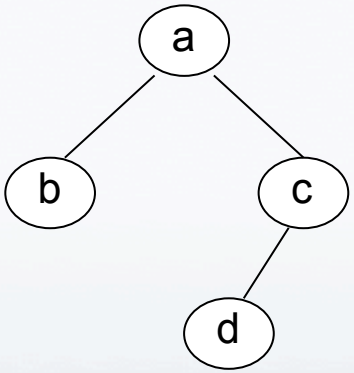
a full tree of depth 3 has 1+2+4+8=15 nodes

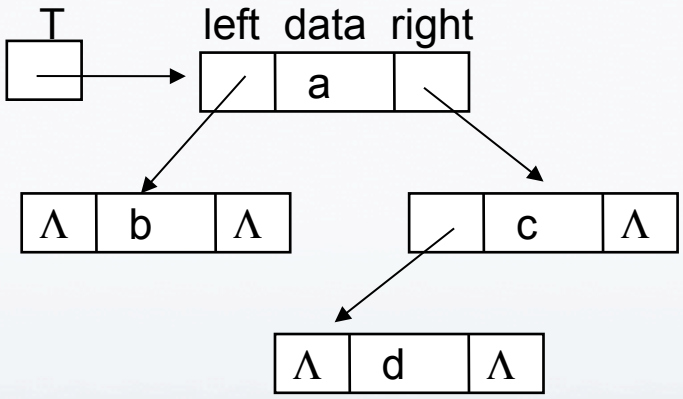
a full tree of depth N has 2^{N+1}-1 nodes

a complete tree of depth N has $2^N-1<m\leq 2^{N+1}-1$ nodes

exercise: what is the depth of a complete tree with m nodes?

DATA STRUCTURES AND REPRESENTATIONS OF TREES





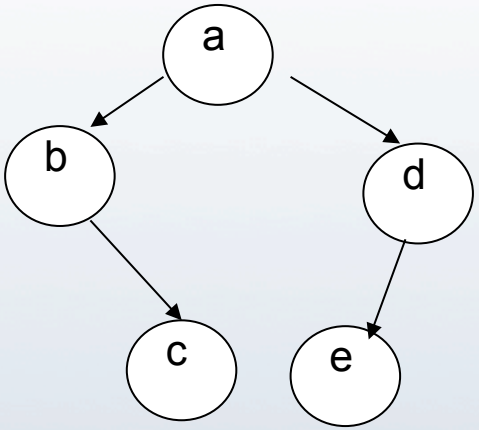

```
template <class T>
class  TreeNode
{private:
TreeNode<T> *left;
TreeNode<T>  *right;
public:
T data;
//constructor
     TreeNode(const T &item, TreeNode<T> *lptr=NULL, TreeNode<T>
                                        *rptr=NULL);
//access methods for the pointer fields
     TreeNode<T>* Left(void)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

//a function 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;
}
```

Example

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

results in:




```
// a function to deallocate memory template <class T>
void FreeTreeNode(TreeNode <T> *p)
{delete p;}
```

Tree Traversal Algorithms:

1: DEPTH-FIRST:

Inorder:
1. Traverse left subtree
2. Visit node (i.e. process node)
3. Traverse right subtree

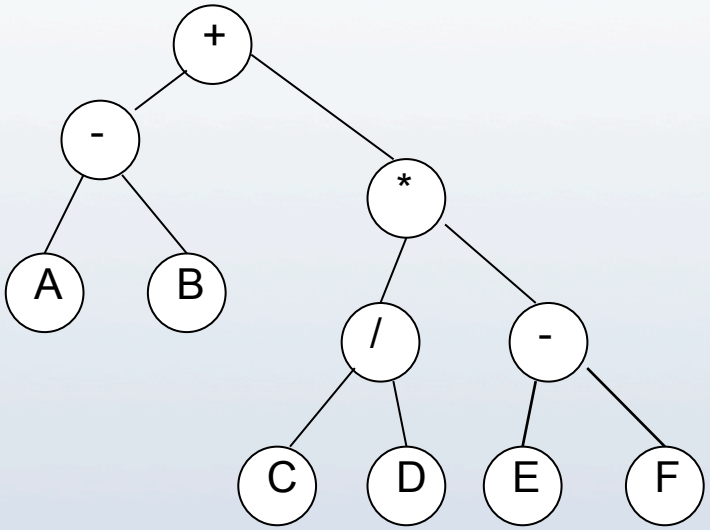
Preorder:
1.Visit node
2. Traverse left
3. Traverse right

Post-order:
1. Traverse left
2. Traverse right
3. Visit node

```
template <class T>
void Inorder (TreeNode<T>* t>
{if (t!=NULL)
{Inorder(t->left)
Cout<< t->data
Inorder(t->right)
}
```

Example:

An arithmetic expression tree stores operands in leafs, operators in non-leaf nodes:



inorder traversal: (LNR)

(A-B)+((C/D)*(E-F))

A-B+C/D*E-F (paranthesis assumed)

postorder traversal: (LRN)

AB-CD/EF-*+

preorder traversal: (NLR)

+-AB*/CD-EF

Note: Postorder traversal, with the following implementation of visit :

if operand PUSH

if operator POP two operands, calculate, push result back

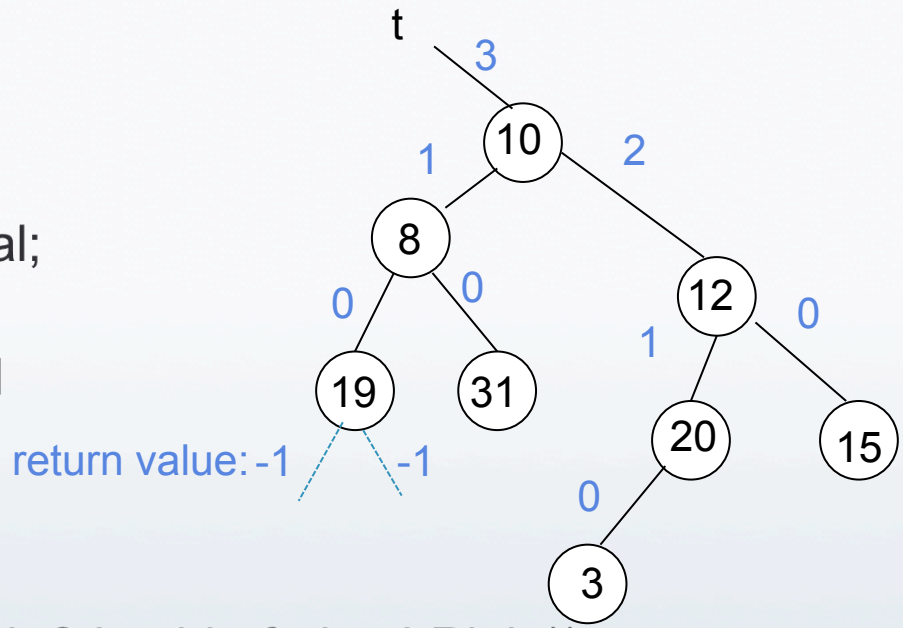
corresponds to arithmetic expression evaluation.

Example: Counting leaves in a tree

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

template <class T>
int CountLeaf ( TreeNode <T> *t)
{
  int countmytree=0;
  CountLeafR(mytree, countmytree);
  return Countmytree;
}
```

Example: computing depth of a tree

```
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?depthLeft:depthRight));
}
 return depthval;
}
```

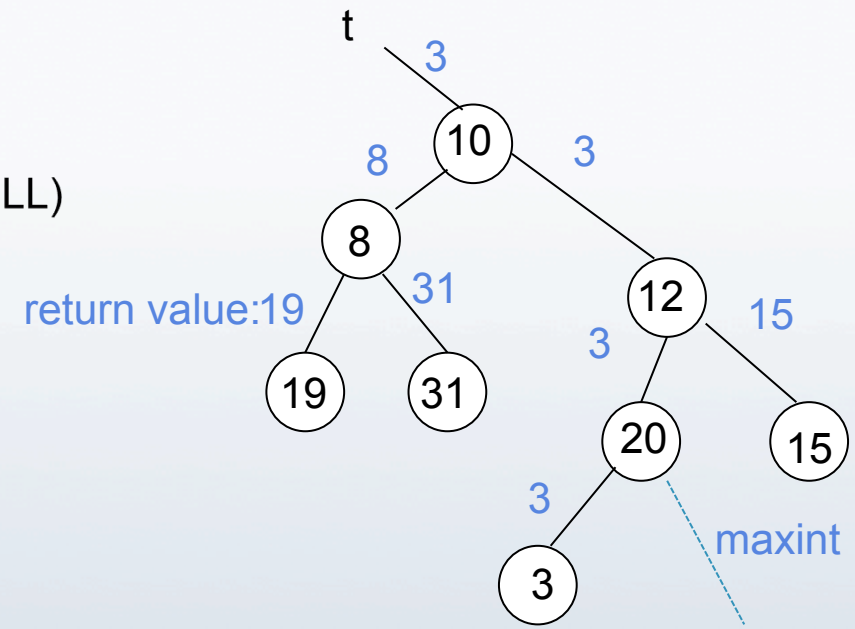


conditional expression syntax:
CONDITION? True-case-EXP:False-case-EXP

The preorder, postorder and inorder traversals are "depth-first"

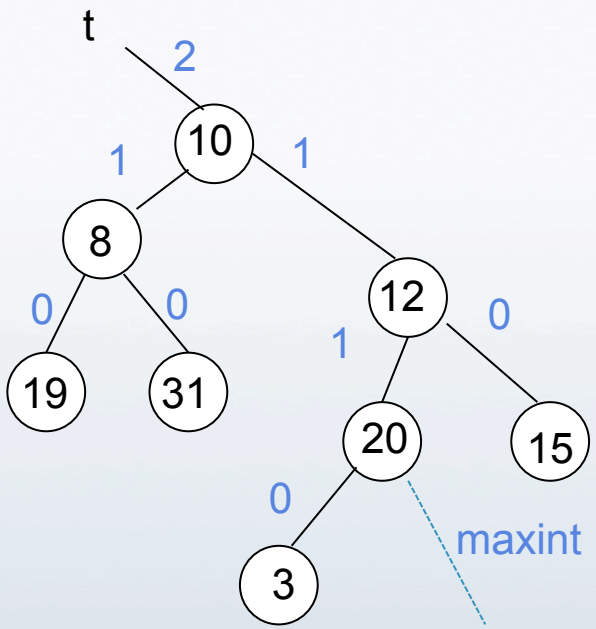
Example: Find min key value stored in binary tree pointed by t.
(assume tree is not empty)

```
template <class T>
int minval ( TreeNode <T> *t)
{
  int minval;
  if( t-> left ==NULL)&& (t->right == NULL)
      return t->data;
  else
  {
    if ( t->left != NULL)
        minleft = minval(t->left)
    else
        minleft = maxint;
    if ( t->right != NULL)
        minright = minval(t->right)
    else
        minright = maxint;

    minval=(minleft < minright? minleft :minright);
    minval = (minval < t->data ? minval : t->data);
    return minval;
  }
}
```



Example: Find the level of the leaf node at minimum level.

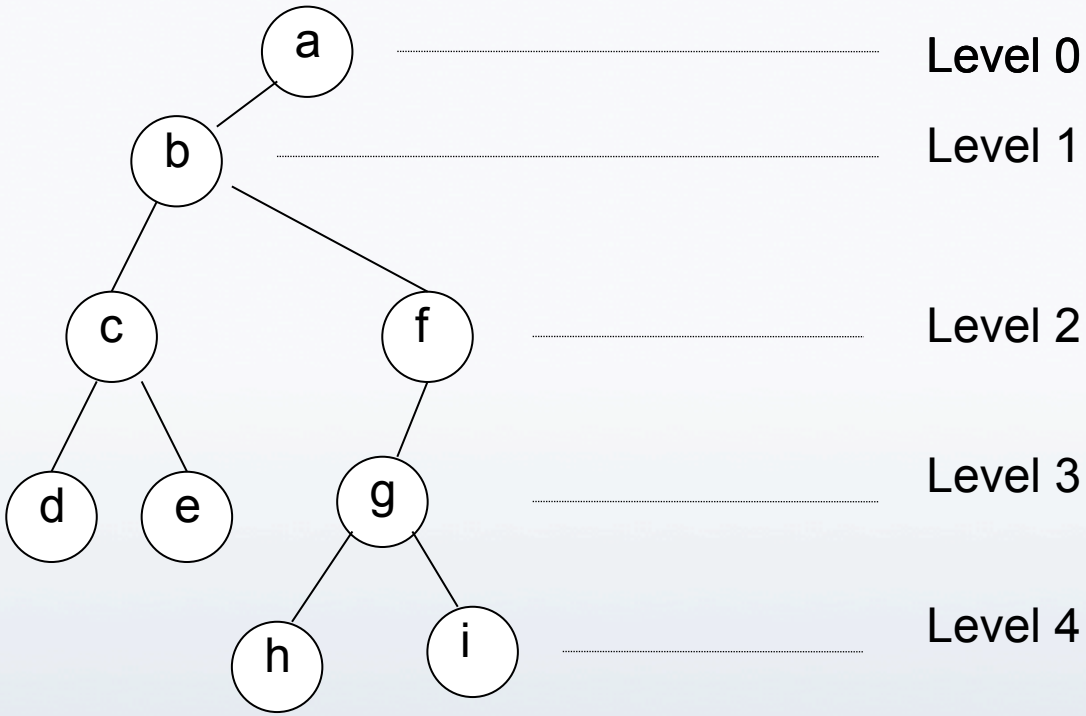
```
template <class T>
int minLeafDepth( TreeNode <T> *t)
{
  if (t==NULL)
    return -1

  int levelleft, levelright, level;

  if (t->left ==NULL && t->right == NULL)
    return 0
  else {
    if (t->left != NULL)
      levelleft = minLeafDepth(t->left)
    else
      levelleft = maxint;
    if (t->right != NULL)
      levelright = minLeafDepth(t->right)
    else
      levelright = maxint;

    level = (levelleft < levelright ? levelleft : levelright) + 1;
    return level;
  }
}
```



Algorithm Level-Traverse (Breath First)

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

```
template <class T>
void LevelScan(TreeNode<T> *t)
{
queue<TreeNode<T>*> Q; // queue of pointers
TreeNode<T> *p;
// insert root
Q.Qinsert(t);
while(!Q.Empty())
{P=Q.Qdelete();
visit(P->data); //for ex. Cout P->data
if (p->Left()!NULL) Q.Qinsert(p->Left());
if (p->Right()!=NULL) Q.Qinser(p->Right());
}
}
```



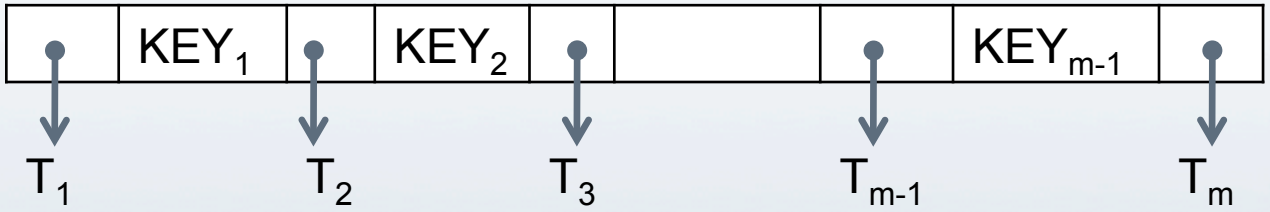
Breath first traversal result:

abcfdeghi

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) $key_1 < key_2 < ... < key_{(m-1)}$

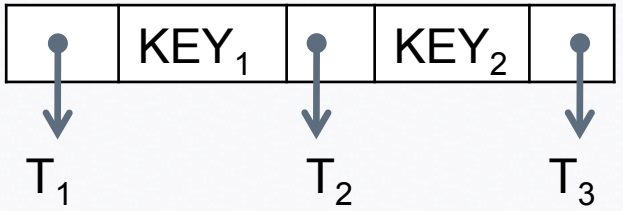
in other words, $key_i < key_{(i+1)}$, $1 <= i < m-1$

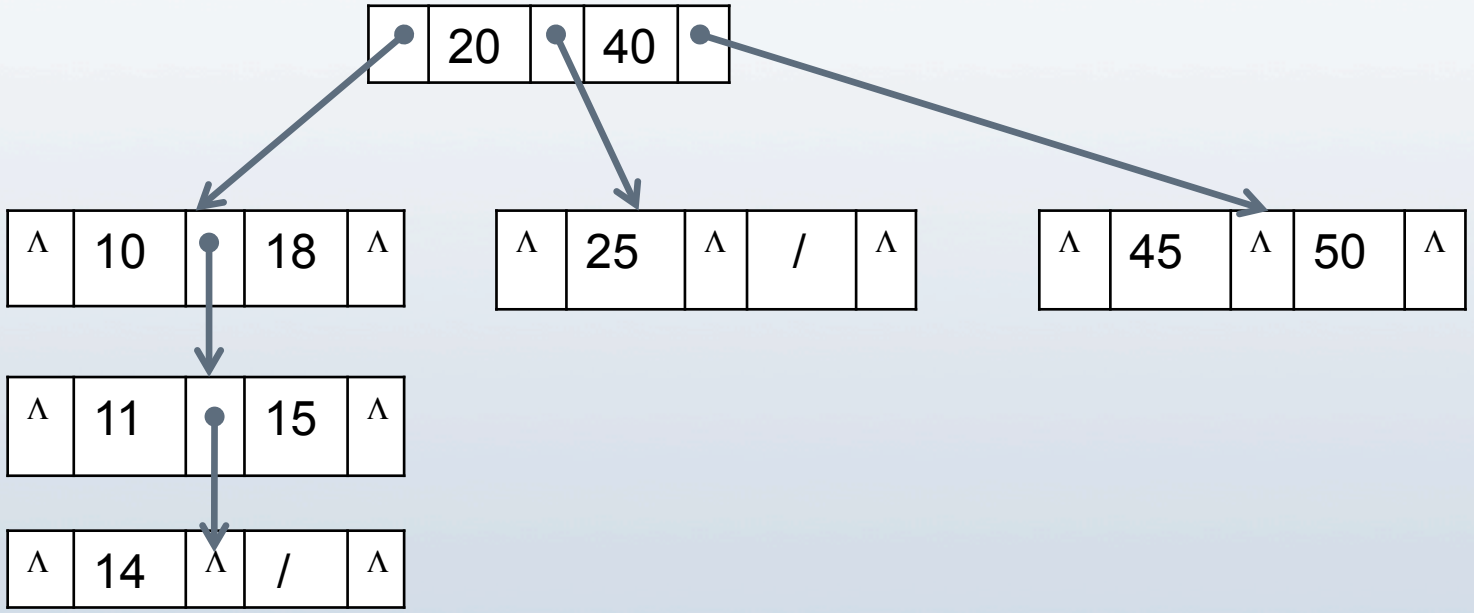
c) All Key values in subtree T_i are greater than Key_i and less than Key_{i+1}

Sometimes, we have an additional entry at the leftmost field of every node, indicating the number of nonempty key values in that node.

Example: 3_way search tree:

Nodes will be of type:





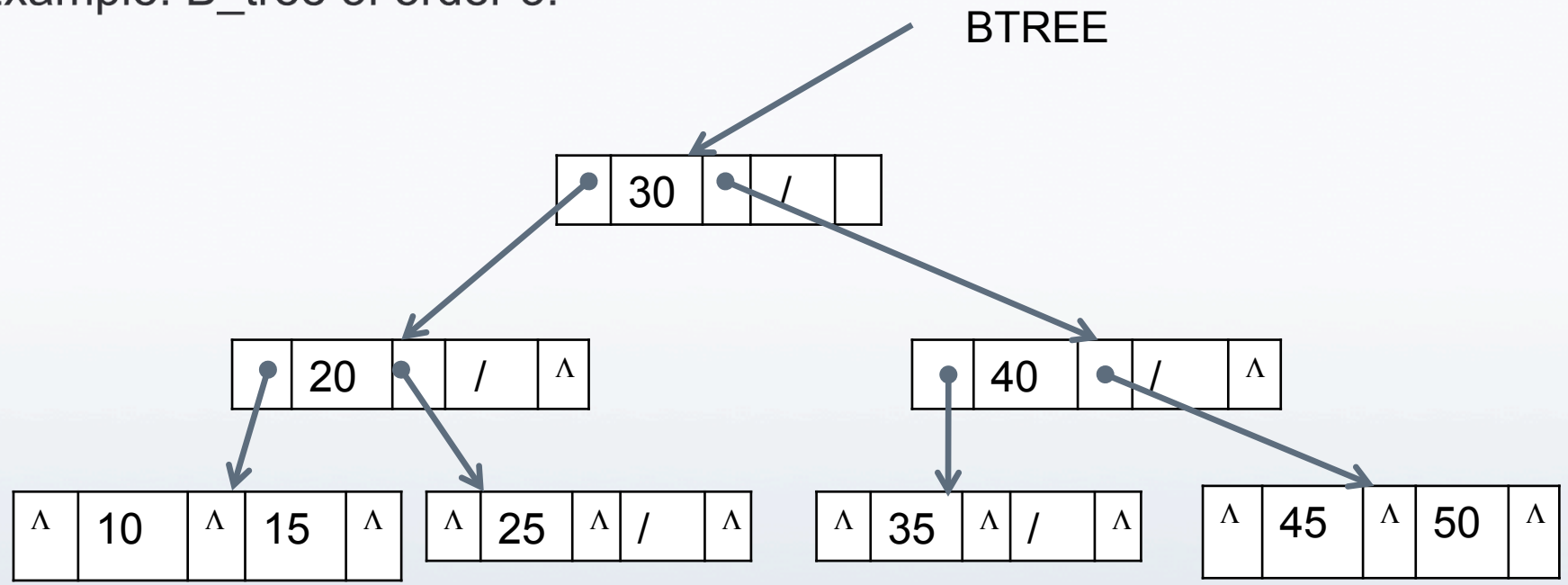
B_TREES

A B_tree of order m is an m_way search tree (possibly empty) satisfying the following properties (if it is not empty)

a) All nodes other than the root node and leaf nodes have at least $\lceil m/2 \rceil$ children,

a) The tree is balanced. (all the leaf nodes are at the same level)

Example: B_tree of order 3:



Inserting new key values to B_tree:

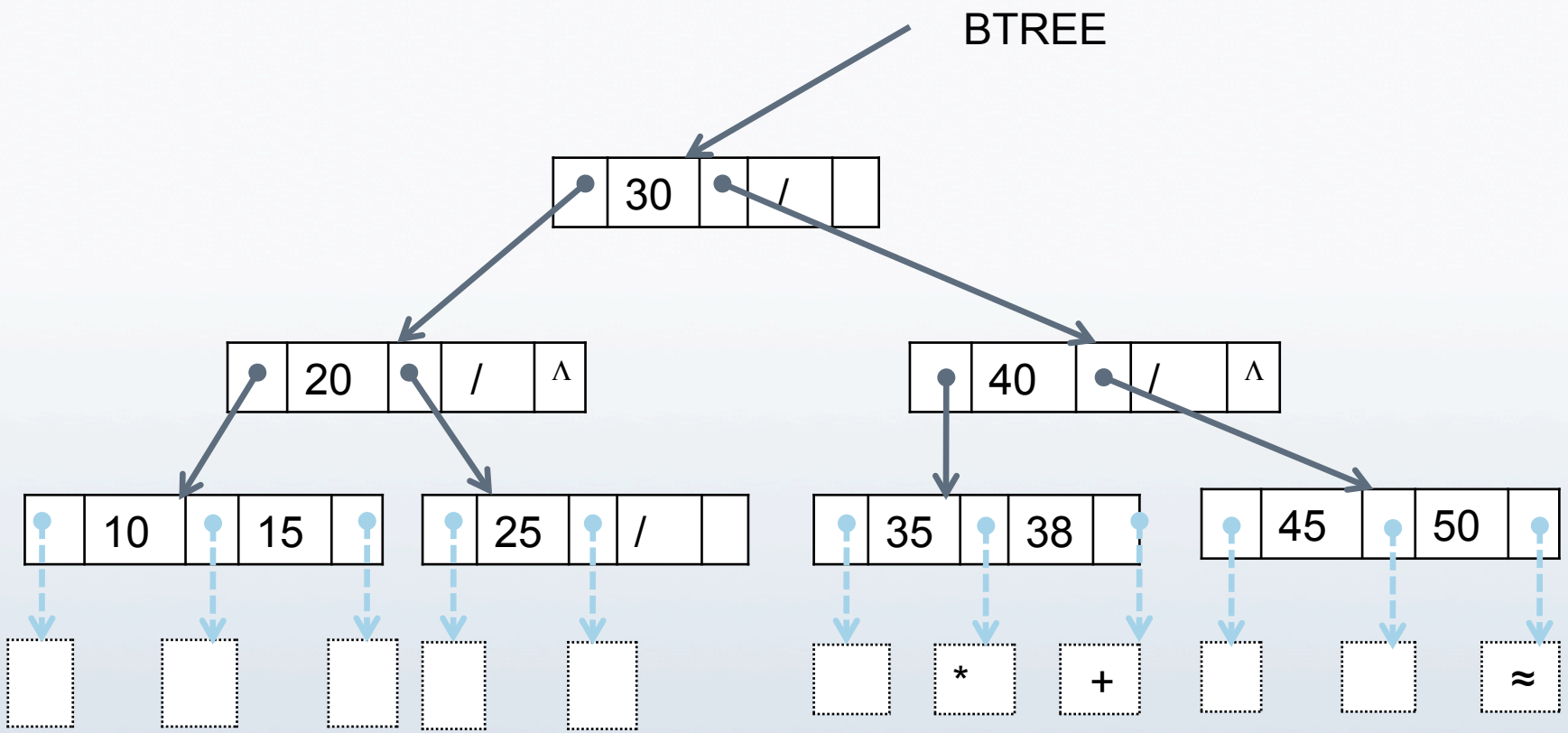
- We want to insert a new key value: x, into a B-tree
- The resulting tree must also be a B-tree. (It must be balanced.)
- We'll always insert at the leaf nodes.

Example:

1) Insert 38 to the B_tree of the above example:

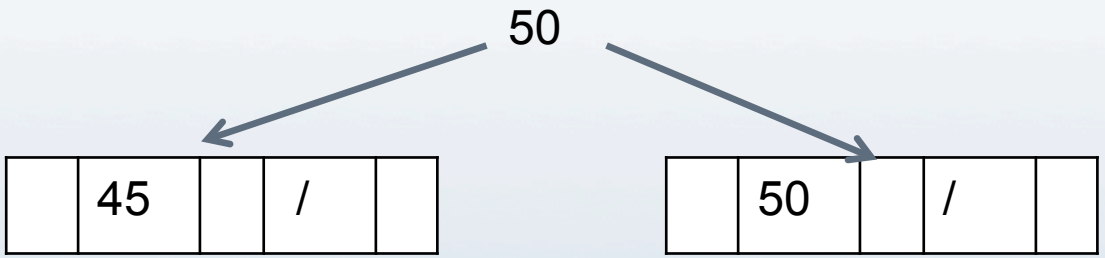
(imagine that we modify all link fields of leaf nodes to point to special nodes called failure nodes)

First of all, we do a search for 38 in the given B_tree. we hit the failure node marked with "*" . The parent of that failure node has only one key value so it has space for another one. Insert 38 there , add a new failure node, which is marked as "+" in the following figure, and return.


BTREE
30
/
20
/
Λ
40
/
Λ
10
15
25
/
35
38
45
50
*
+
≈

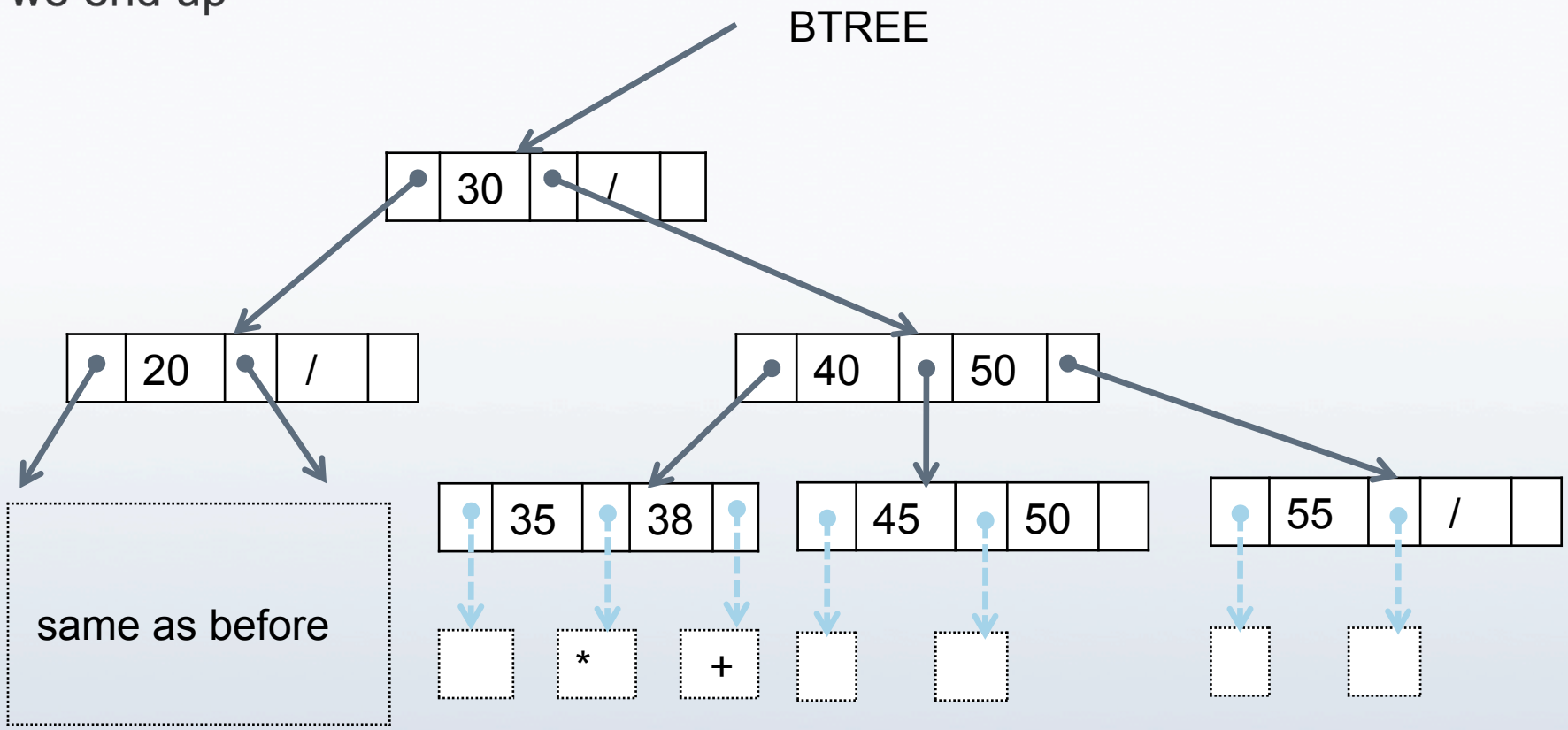
2) Now, insert 55 to this B_tree. We do the search and hit the failure node "≈". However, it's parent node does not have any space for a key value. Now, assume we create a new node instead of

and split into two nodes

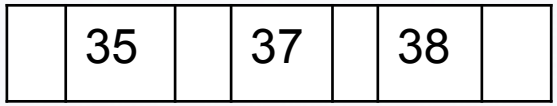


and inset 50 to parent node

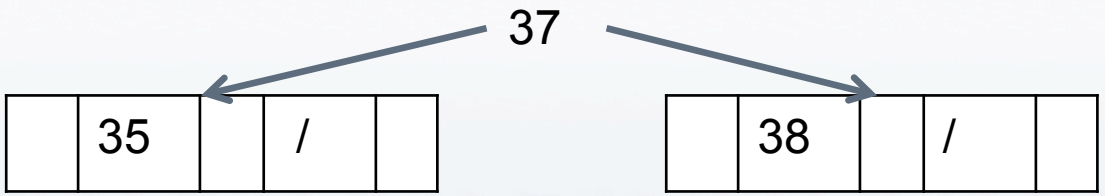
so we end up



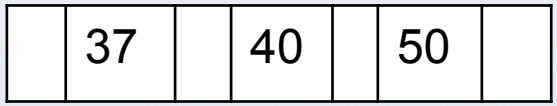
3) Now, let us insert 37 to this B_tree: We search for 37, and hit a failure node between 35 and 38. So,we have to create:



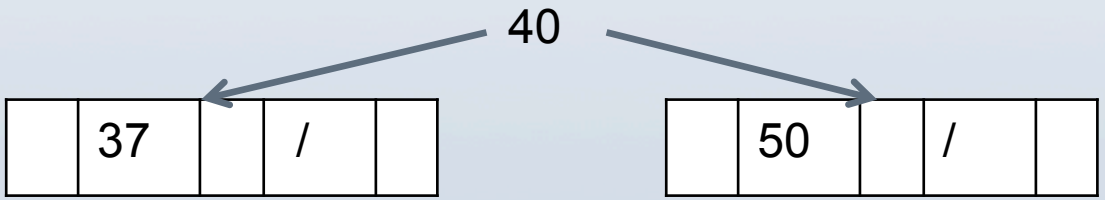
split it



and inset 37 to its parent

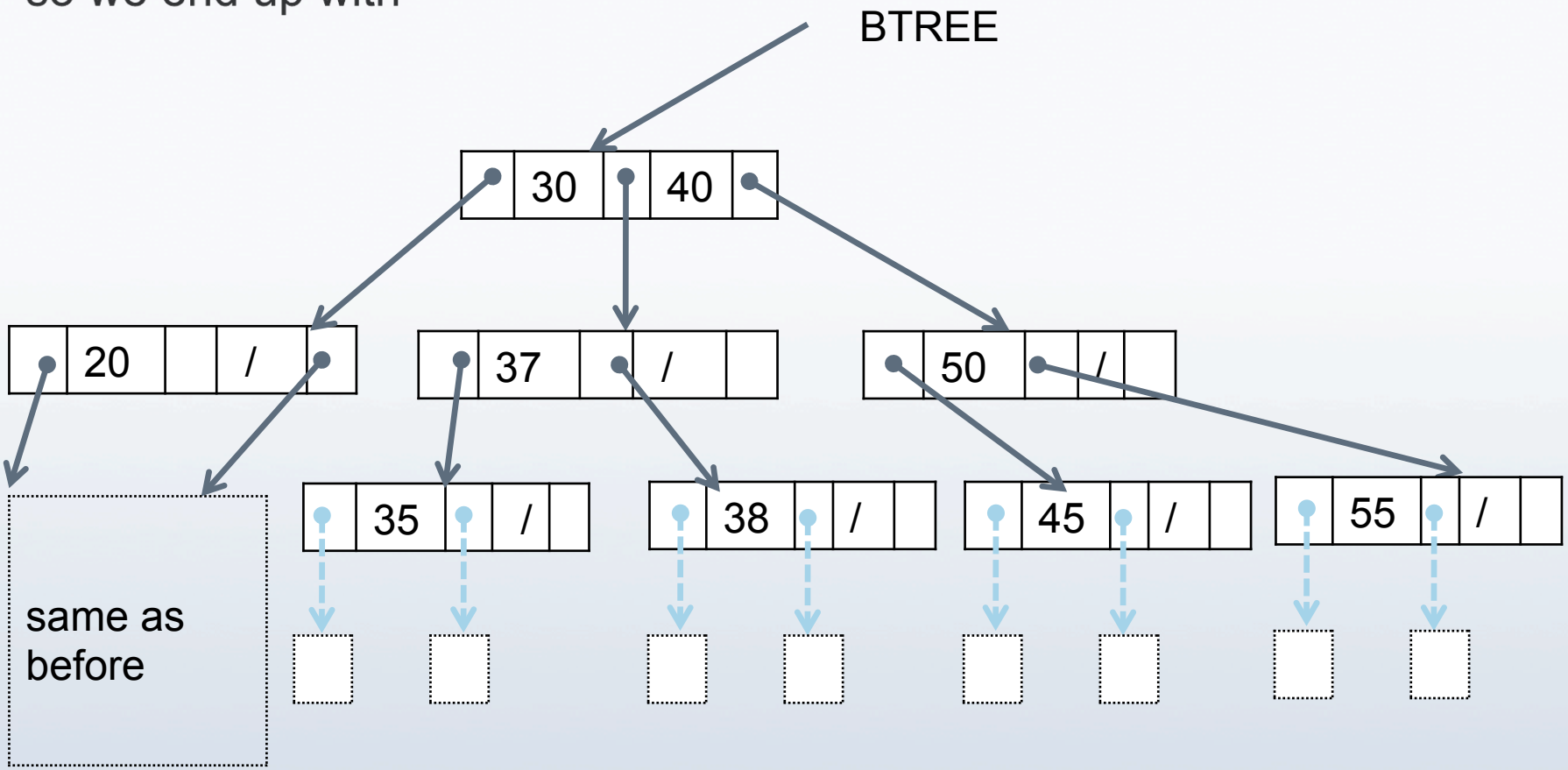


since there is no space for a new key, split it again



and this time inset 40 into its parent

so we end up with



- If we can not insert to the root node, we split and create a new root node. For example try to insert 34, 32, and 33. Thu, the height of the B_tree increases by 1 in such a case.
- Deletion algorithm is much more complicated! It will not be considered here.