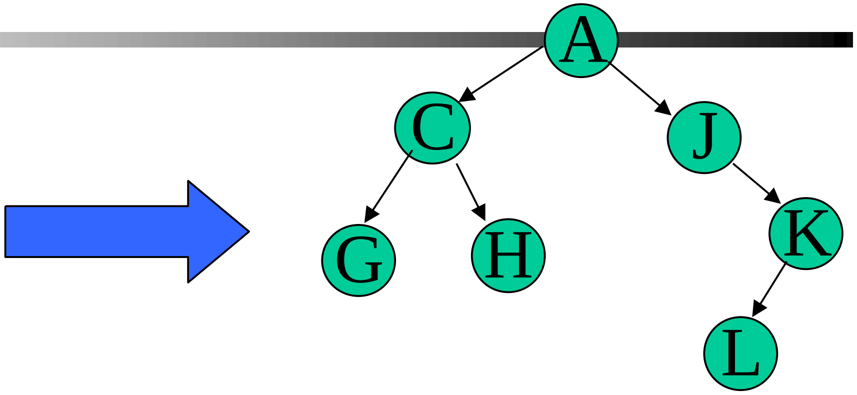
Trees - I

Overview

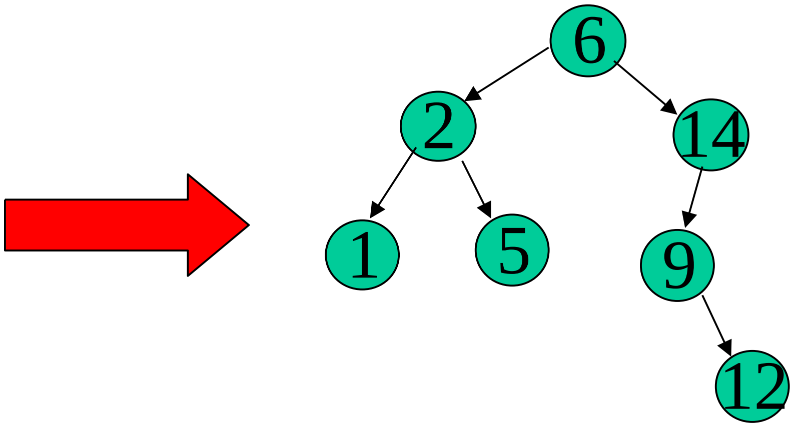
- Binary trees
  - Definition, Representation & implementation.
  - Binary search trees –construction, search and deletion.
- Tree traversals
  - inorder, preorder and postorder.
- Tree types
  - Search Trees, Expression trees, Degenerated Tree.
- Polish Notation

Binary Tree

- An N-ary tree with at most two sub-trees
  – a left subtree (LST)
  – a right subtree (RST)

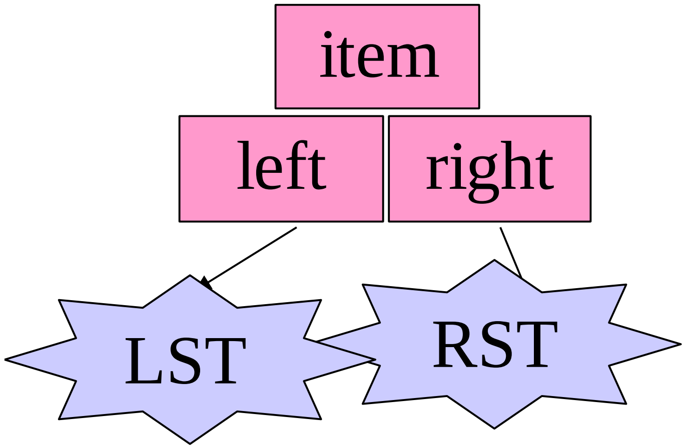


If value of LST, value of node and value of RST are in strict order for all nodes in the tree, we have a **Binary Search Tree**



BST Implementation

```
typedef struct TreeNode{
  int item;
  TreeNode *left;     //reference to LST
  TreeNode *right;    //reference to RST
};
```



Implementation

- Variations used depending on application - We will ignore variations largely.
- We need a class to hold the tree and support operations such as–
  - insert an element
  - delete an element
  - search an element
  - traversal.

Implementation…

```
class BinaryTree {
 private:
        TreeNode* root;
        TreeNode* search(TreeNode* root, int key);
 public:
       BinaryTree (){root=NULL;}
       bool is_empty(){return (root==NULL);}
       TreeNode* search(int key){
                return search(root,key);
              }
```

Implementation…

```
void insert_item(int item) { // }
void preorder() {preorder(root);}
void inorder() {inorder(root);}
void postorder() {postorder(root);}
void delete_item(int item) { // }
};
```

Searching in BST

Looking at a node, we know whether to proceed left or right.

```
TreeNode* search(TreeNode* p, int key){
  if (p != NULL){
   if(key== p->item) return p; //Found
   else if (key < p->item)
return search(p->left, key);
   else
return search(p->right, key); }
  return p; }
```

Traversal

- Means visiting all nodes of a tree, in some order, systematically.
- In general many traversals are possible (= n!, n is the number of nodes in the tree)
- Must ensure that all nodes are visited, once and only once.
- Two common ways to traverse the nodes
  - Breadth-first traversal (BFT)
    - Visiting all nodes at particular level before next level.
  - Depth-first traversal (DFT)
    - Traverse one sub-tree fully before moving to another.

Traversal

- As per the definition, there are three components at a node of a tree.
- Three different traversals commonly followed - PRE, IN, POST order decided by the order of visiting these components.
- We assume LST is visited before RST anyway - option decided by when to visit root.

Preorder Traversal

```
void preorder(TreeNode* root){
  if (root != NULL){
    //access root->item;
    preorder(root->left);
    preorder(root->right);
  }
}
```

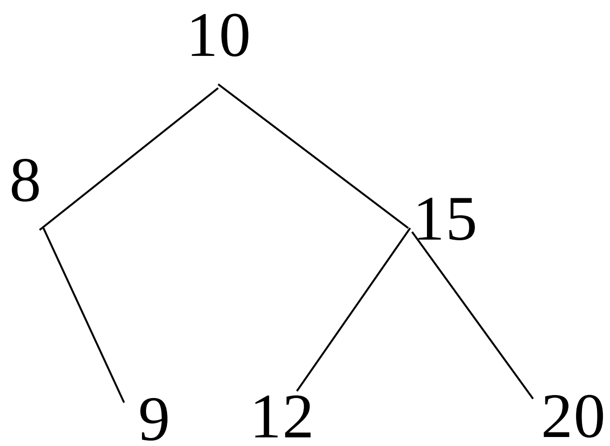
Inorder Traversal

```
void inorder(TreeNode* root){
   if (root != NULL){
      inorder(root->left);
      //access root->item;
      inorder(root->right);
   }
}
```

Postorder Traversal

```
void postorder(TreeNode* root){
   if (root != NULL){
      postorder(root->left);
      postorder(root->right);
      //access root->item
   }
}
```

Traversal of a Search Tree



Pre : 10 8 9 15 12 20

Post: 9 8 12 20 15 10

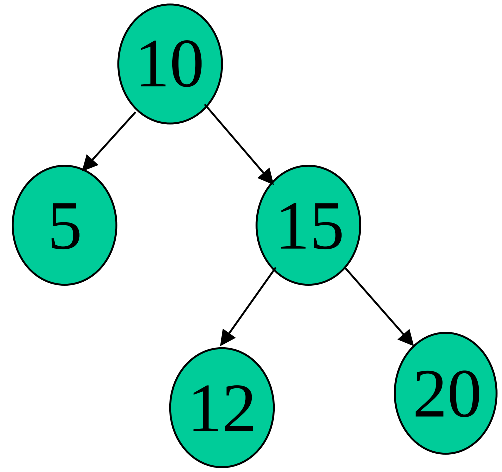
In : 8 9 10 12 15 20

- Inorder traversal generates sorted order!

Constructing BST

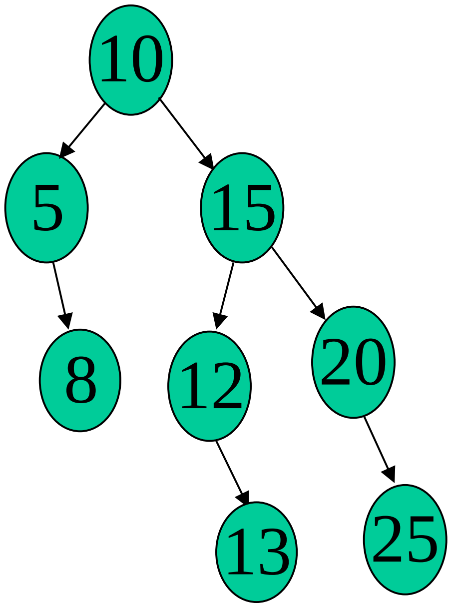
- If tree is empty, make the new node root of the tree.
- If tree is not empty, Search in the existing tree for the element to be inserted.
- We will reach a leaf node - the element is to be added as the left or right child of that node.
- Create that child and put the element there.
  - growth is always at the leaves.

Insert 8, 13, 25



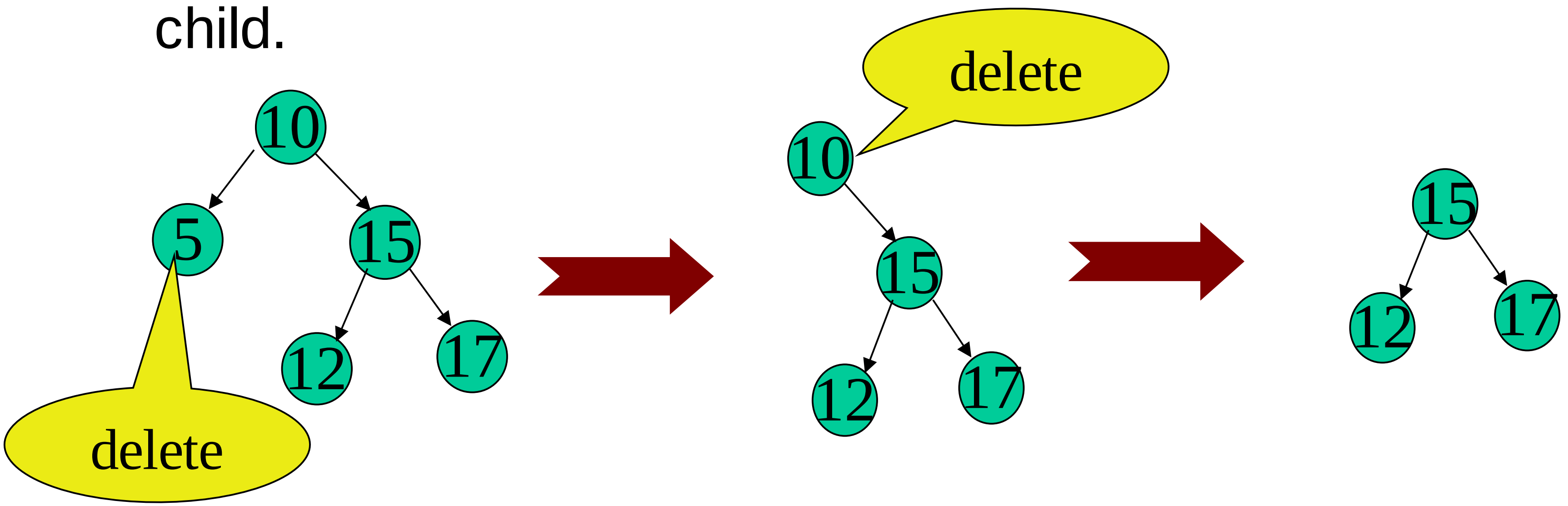
Constructing BST

```
void insert_item(int  item){
  TreeNode* p = root, prev = NULL;
  if(root == NULL) { root = new TreeNode; root->item=item; root->left=NULL;
          root->right=NULL;return;}
  while(p!=NULL){//  find a place for inserting new node
          prev = p;
          if(p->item<item) p = p->right; else p = p->left;}
  if(prev->item<item){
      TreeNode* node=new TreeNode; node->item=item;
      node->left=NULL;node->right=NULL;  prev->right=node;}
  else if(prev->item>item){
      TreeNode* node=new TreeNode;node->item=item;
   node->left=NULL;node->right=NULL;  prev->left=node;}
}
```



Deletion in BST

- Deletion has to preserve BST criteria.
- Node is a leaf? Just delete the node, set parent's pointer to NULL.
- Node has a single child? Set parent's pointer to the child.

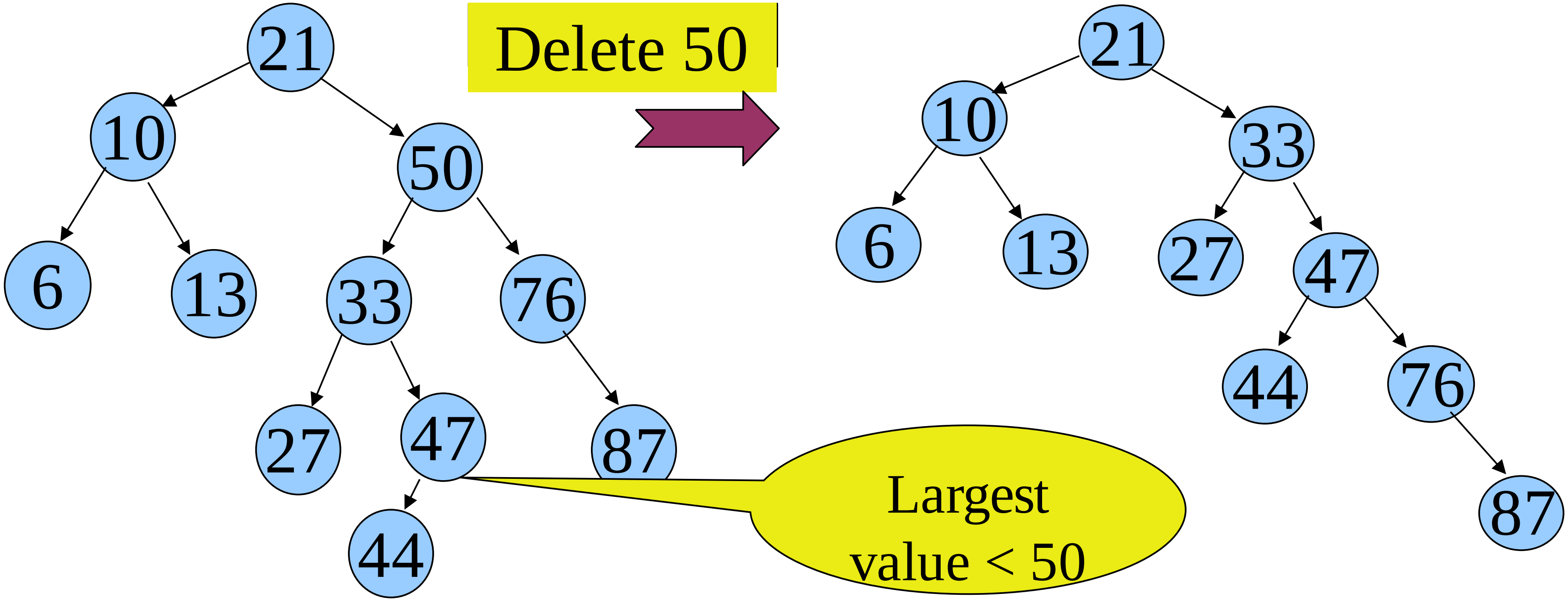


Deletion of Nodes

- Node has both children.
- There are many solutions - must ensure the Search tree property.
- Two ways
  - Deletion by merging
  - Deletion by copying

Deletion by Merging

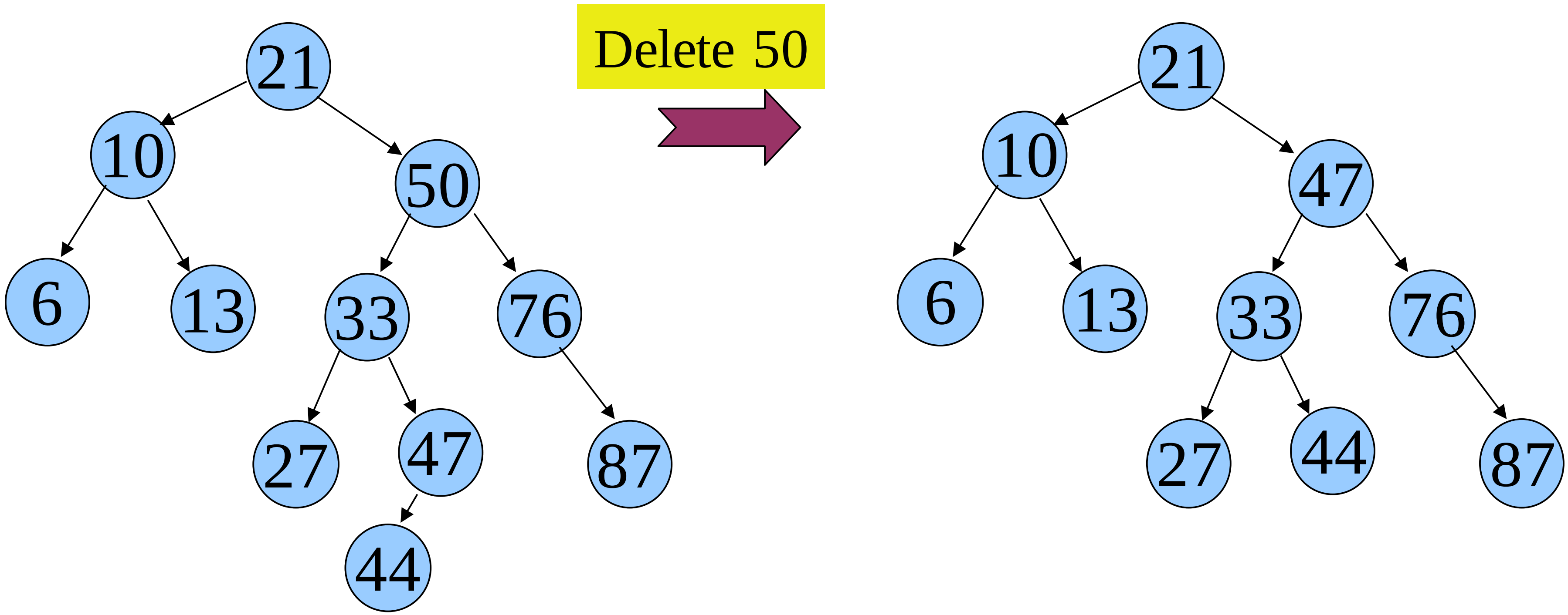
Merge node's RST with its LST and make a single tree by attaching RST as RST of rightmost descendent of LST



Can also merge node's LST with its RST in a similar way.

Deletion by Copying

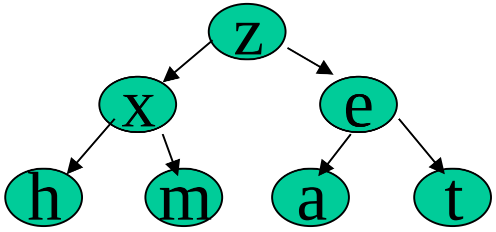
Reducing problem to case 1 or 2 by replacing the node to be deleted with its inorder predecessor(or successor).



Types of Trees

- Expression tree: internal nodes – operators; leaf nodes - operands
- Search tree: info fields of nodes satisfy certain order.
- Complete Binary Tree (CBT): All the internal nodes have both the children and all the leaves are at same level.
- Strictly Binary Tree: Every node has either 0 or 2 children.
- Degenerated tree: Almost like a list.

What could be an input sequence for degenerated binary search tree?



Polish Notation

- Unambiguous binary expression representation by removing parenthesis from it.
- Used by compiler/interpreters for expression evaluation.
- Two types
  - Prefix notation - Operator precedes the operands.
  - Postfix notation - Operator succeeds the operands.

Polish Notation…

- The traditional form is called infix (a+b) * (c+d)

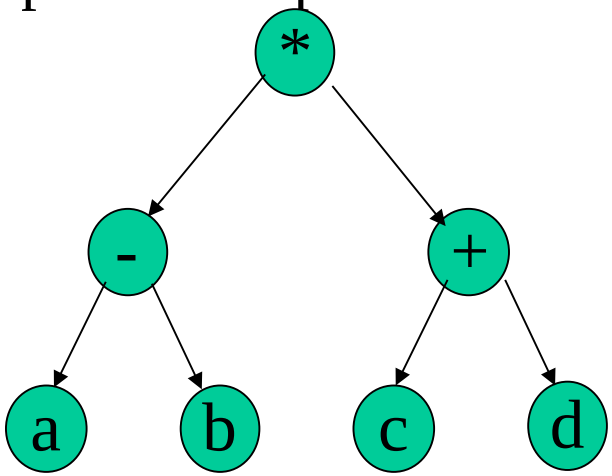
Infix: `(a+b)*(c+d)`

Prefix: `*+ab+cd`

Postfix: `ab+cd+*`

Traversal of Expression Tree

- Represents a parenthesis-free expression.



Preorder: * - a b + c d

Postorder: a b - c d +*

Inorder: a - b * c + d

Problem?

- We get prefix, postfix and infix representation of the expression by suitable traversal.

Application of Binary Trees

- Given a **prefix expression, convert it to infix expression**. Use parenthesis to maintain the precedence of operators. Assume only binary operators and single character operands.
- Examples

Prefix	Infix
`+ a b`	`(a + b)`
`+ + a b c`	`((a + b) + c)`
`* + a b + c d`	`(( a + b )*(c + d ))`

Prefix to Infix Conversion

- Convert prefix expression to binary tree form.
- Perform inorder traversal on the binary tree to get the infix expression.
- For expression tree, we know -
  - Internal nodes will be operators
  - Leaf nodes will be operands

Prefix to Binary Tree Form

```
TreeNode* create_expression_tree(){
 TreeNode* root;
 read(character);
 while(character != '\n'){
   if(character is operator){
     root = new Node; root->item=character;
     root->left = create_expression_tree();
     root->right = create_expression_tree();
  }
```

Prefix to Binary Tree Form

```
else if(character is operand){
    TreeNode* node=new Treenode;
    node->item=character;
    node->left=NULL; node->right=NULL;
    return node;}
 }
 return root;
}
```

Summary

- Binary tree – definition and terminologies.
- Binary search trees – construction, searching and deletion.
- Traversals in Binary trees – preorder, inorder, postorder.
- Expression trees – unambiguous representation of expression; prefix & postfix notation