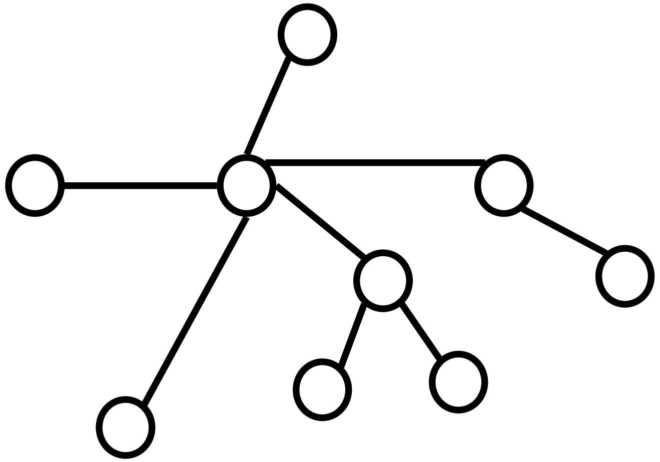
Trees

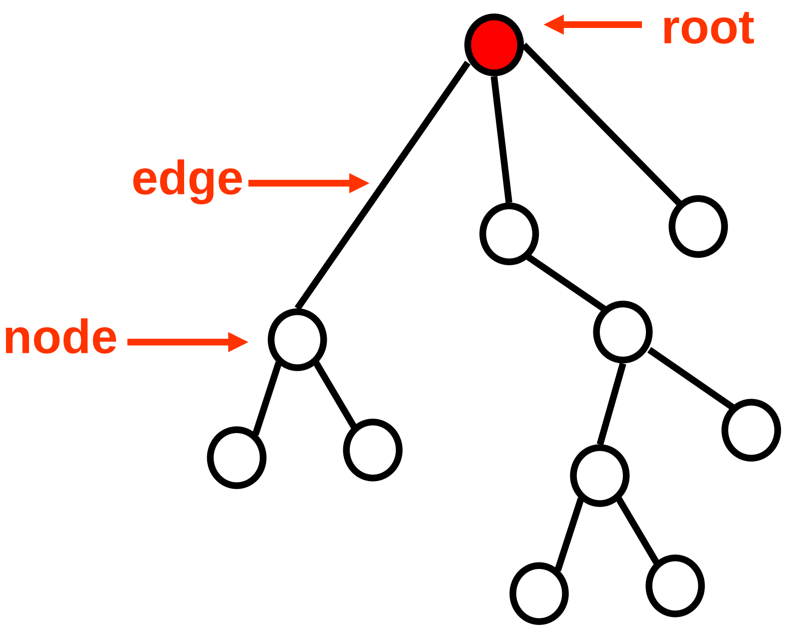
Trees
עצים

- עץ חופשי : אוסף של קודקודים וצלעות אשר **קשורים** ללא מעגלים
- עץ הוא מבנה נתונים לא ליניארי

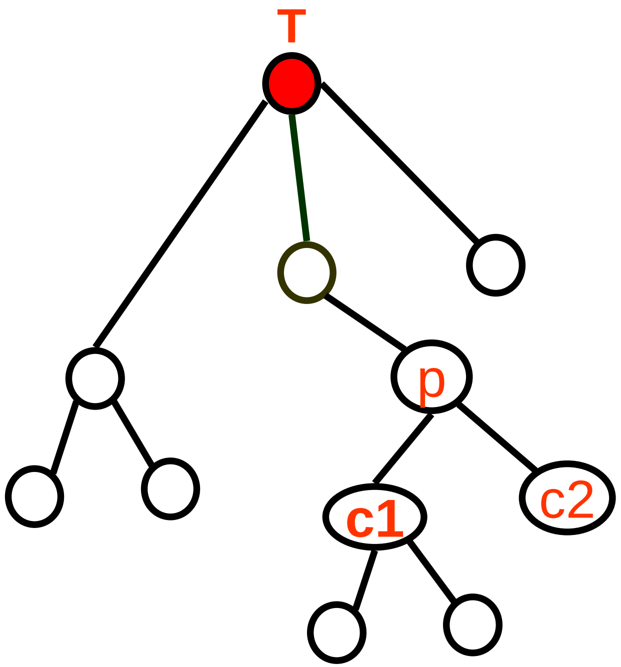


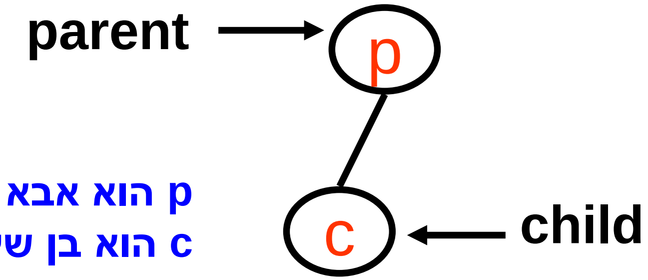
עץ מושרש

- עץ **מושרש** הוא עץ חופשי שבו קיים קודקוד מובחר הנקרא שורש (root)
כל קודקוד בעץ מושרש נקרא צומת (node)

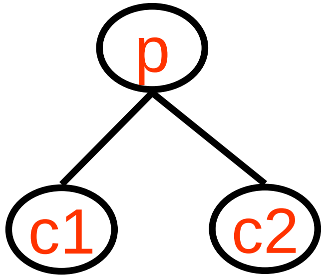


מושגים כלליים





p הוא אבא של c
c הוא בן של p



c1 ו- c2 הם אחים (siblings)
אם יש להם אבא משותף

מושגים כללים
המשך

- אב קדמון ancestor

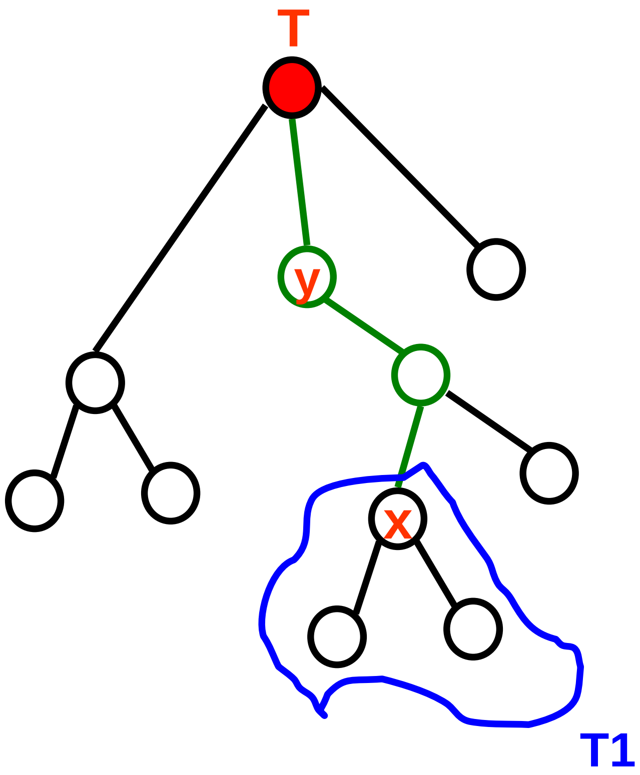
יהי T עץ מושרש , ותהי x צומת כלשהי בעץ המושרש T.
נגיד שצומת y היא אב קדמון של x אם y נמצא על המסלול היחיד מהשורש עד לצומת x .

- צאצא descendant

תהי x צומת כלשהי בעץ מושרש T
נגיד ש x הוא צאצא של y אם y הוא אב קדמון של x

- תת עץ sub tree

T1 תת עץ המושרש ע"י צומת כלשהי (נניח x) הינו העץ הנוצר ע"י צאצאיו של x ואשר x הוא השורש שלו



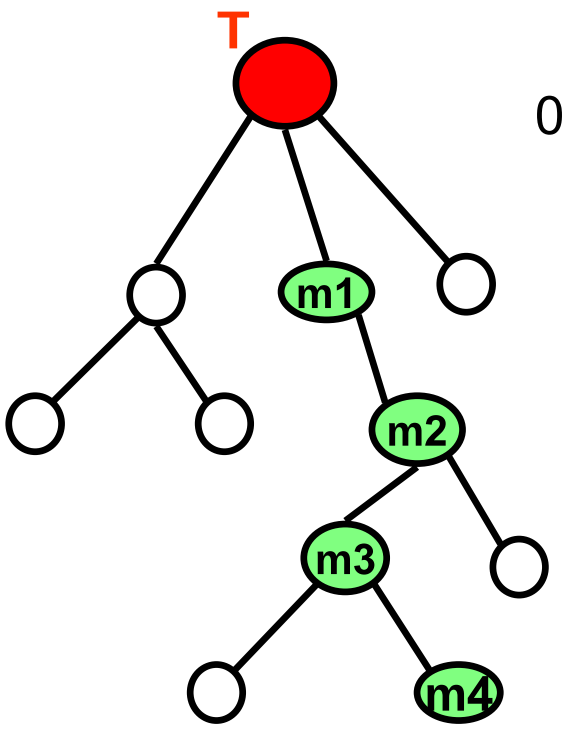
מושגים כללים
המשך

- יהיו m1,m2,m3…mk רצף של קודקודים כך ש m1 הוא אבא של m2 וזה אבא של m3 וכן הלאה.
- נגדיר את m1,m2,…mk להיות המסלול (path) מ m1 ל- mk
- אורך המסלול (path length) של m1,m2….mk הוא k-1

שאלה: איזה מסלול בעל אורך 0?

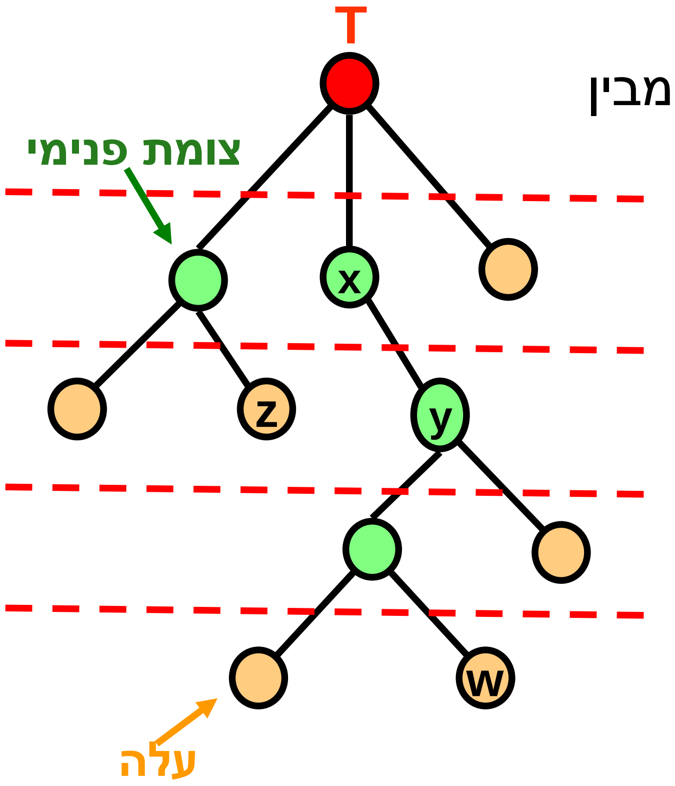
תשובה: כלומר מסלול המורכב מקודקוד בודד הוא בעל אורך 0 כאשר k=1 אז אורך המסלול הוא אפס

- גובה height של קודקוד x כלשהו הוא אורך המסלול הארוך ביותר מ x עד לעלה שלו(מספר הצלעות עד לעלה)
- גובה עץ הוא גובה השורש (עד לעלה העמוק ביותר)



מושגים כללים
המשך

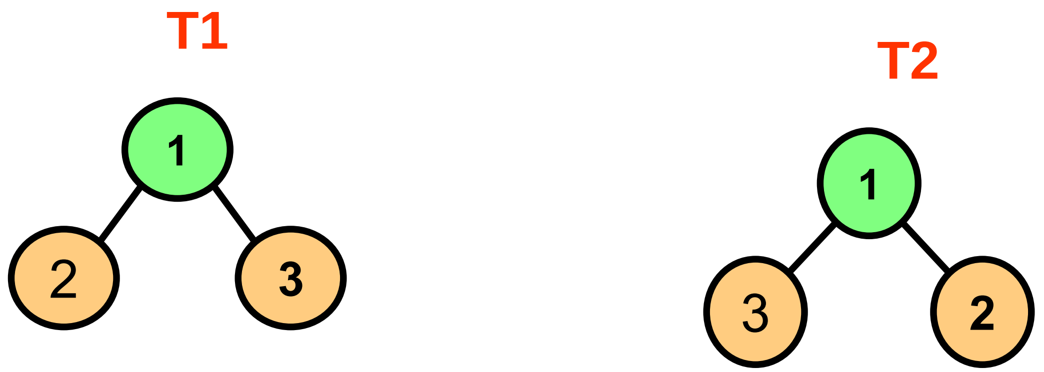
- עלה leaf הינו קודקוד (צומת) שאין לו בנים
- קודקוד פנימי internal node קודקוד שאינו עלה
- דרגה degree של צומת כלשהי היא מספר הבנים של הצומת
- עומק depth / level של צומת כלשהי הוא אורך המסלול (מס צלעות) מן השורש עד הצומת (root level = 0)
- גובה height של עץ T כלשהו הוא העומק הגדול ביותר מבין כל העומקים של כל הצמתים בעץ



עומק של צומת x 🡐 1
עומק של צומת y 🡐 2
עומק של צומת z 🡐 2
עומק של צומת w 🡐 4
גובה העץ T 🡐 4

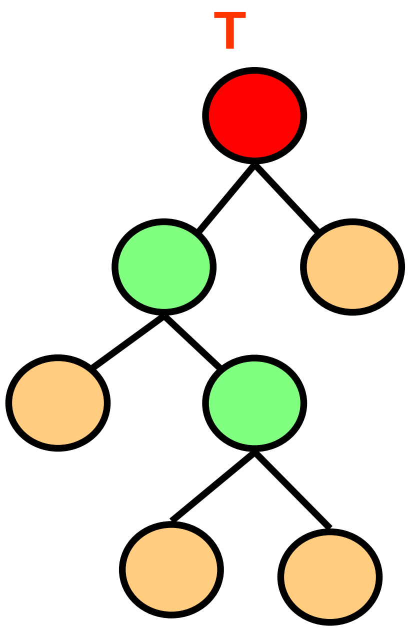
Ordered Tree
עץ סדור

- עץ סדור הינו עץ מושרש שבו הבנים הם סדורים



T1 ו- T2 זהים כעצים מושרשים אך שונים כעצים סדורים

Binary Tree
עץ בינארי

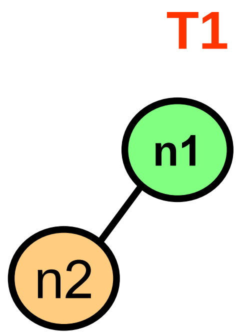


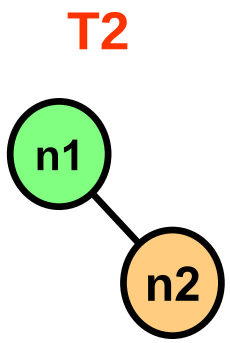
- עץ בינארי הוא עץ מושרש המקיים :

I. עץ ריק (אינו מכיל צמתים) או
II. עץ המורכב משלוש קבוצות של צמתים :
   1. ראש
   2. תת עץ שמאלי (גם הוא בינארי)
   3. תת עץ ימני (גם הוא בינארי)

- לכל צומת יש לכל היותר שני בנים (שמאלי וימני)
- בעץ בינארי יש חשיבות למיקום הבן (שמאלי או ימני)
- דרגה degree של כל צומת בעץ בינארי היא 0 , 1 או 2

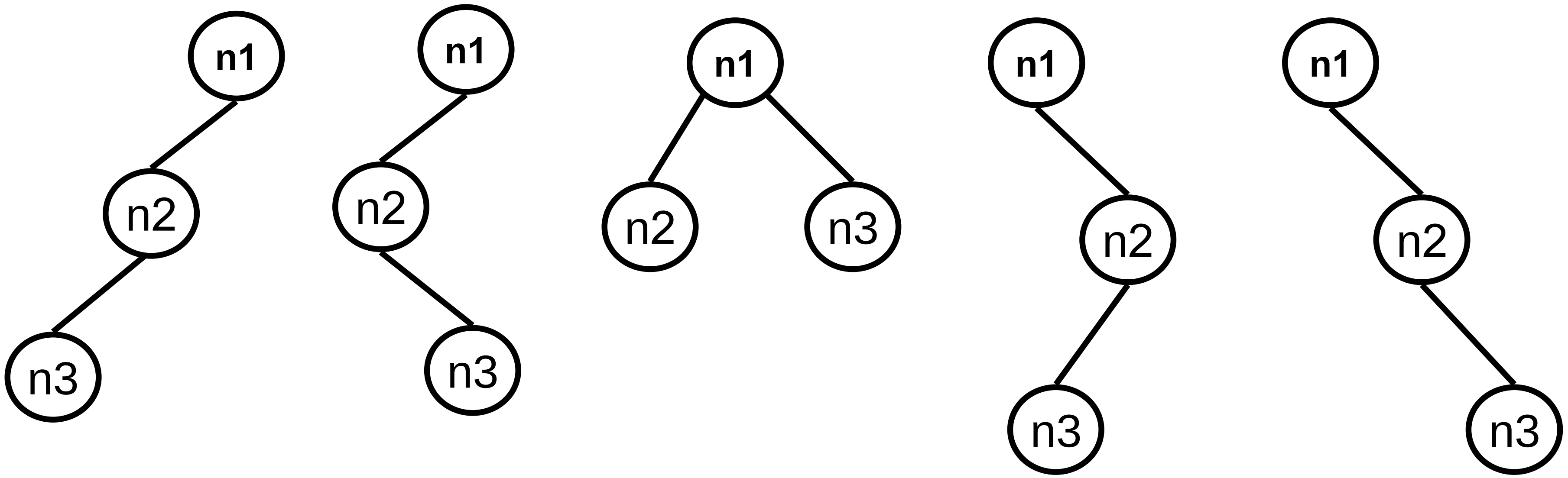
Binary Tree
עץ בינארי





T1 ו- T2 הם עצים בינאריים שונים (יש חשיבות לסדר של הבן השמאלי והימני)

Binary Tree
עץ בינארי

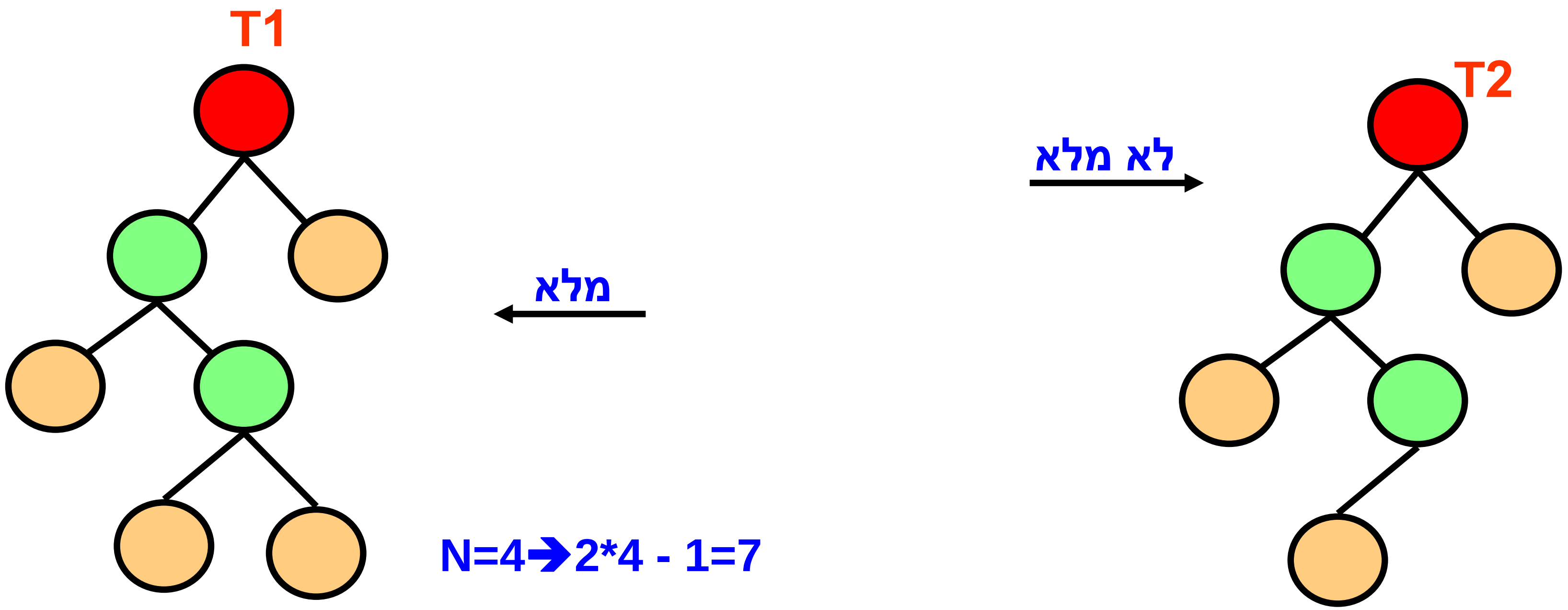


עצים בינאריים שונים (המורכבים מ 3 צמתים)

Full Binary Tree
עץ בינארי מלא

▪עץ בינארי מלא הוא עץ בינארי בו כל הקודקודים פרט לעלים הם בעלי דרגה 2. כלומר לכל צומת או שהוא עלה או שיש לו שני בנים בדיוק כלומר הדרגה של כל צומת או 0 או 2 (אף פעם לא 1).

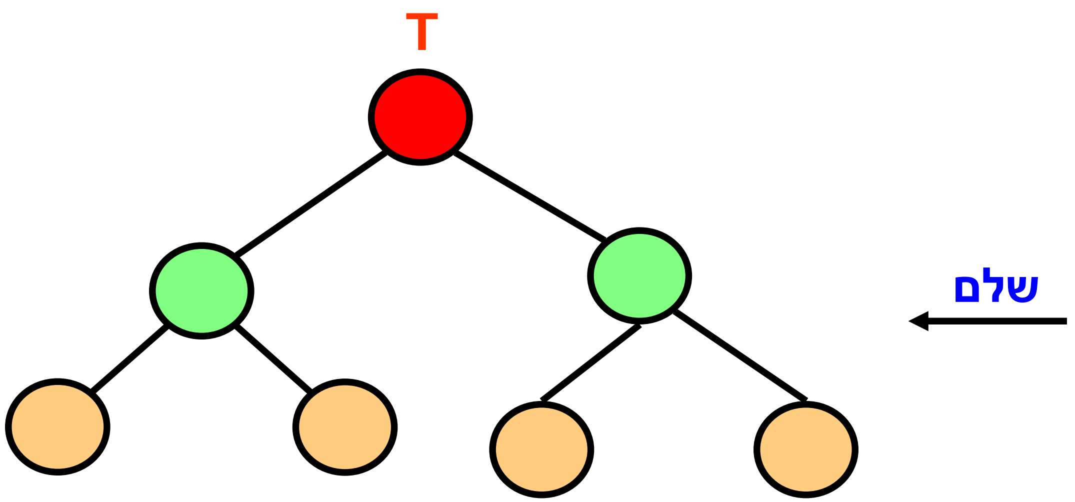
תכונה: בעץ בינארי מלא אם מספר העלים הוא N אז מספר הצמתים הכולל הוא 2N-1



Complete Binary Tree
עץ בינארי שלם

- עץ בינארי שלם הוא עץ בינארי שבו כל העלים באותו עומק ושבו לכל צומת פנימי יש 2 בנים (עץ בינארי מלא שבו כל העלים באותו עומק)
- תכונה: הגובה h של עץ בינארי שלם בעל N צמתים הוא לכל היותר O(logN)
- תכונה : מספר הצמתים n בעץ בינארי שלם ניתן לפי הנוסחה $n = 2^{h+1} - 1$ כאשר h הוא גובה העץ
- תכונה : מספר העלים L בעץ בינארי שלם הוא $L = 2^h$ כאשר h הוא גובה העץ ומספר הצמתים הפנימיים הוא $2^h - 1$

h=0→n=1
h=1→n=3
h=2→n=7



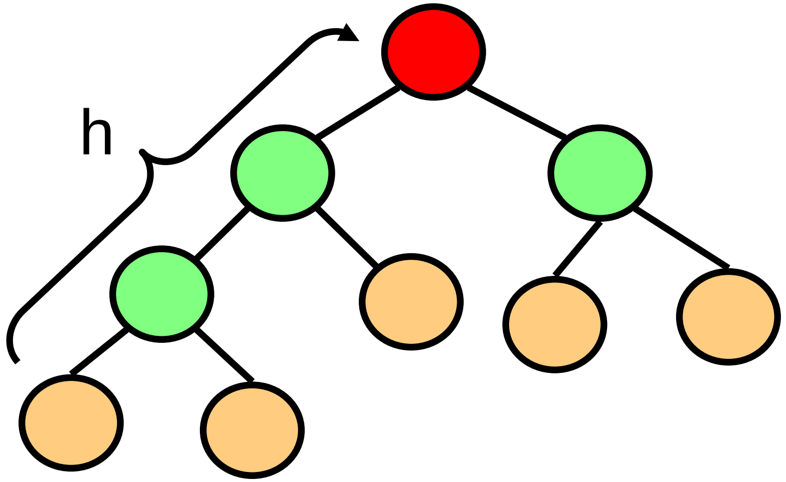
Almost Complete Binary Tree
עץ בינארי כמעט שלם

עץ בינארי כמעט שלם הוא עץ בינארי שבו כל הרמות מלאות חוץ מהרמה התחתונה שבה החל ממקום מסוים יתכן שאין יותר קודקודים

תכונה:מספר הצמתים בעץ בינארי כמעט שלם הוא מ $n=2^h$ עד $n=2^{h+1}-1$

בעץ בינארי כמעט שלם בגובה h כל העלים נמצאים או בעומק h או h-1

גובה של עץ בינארי כמעט שלם בעל n צמתים $h=\lfloor \log n \rfloor$

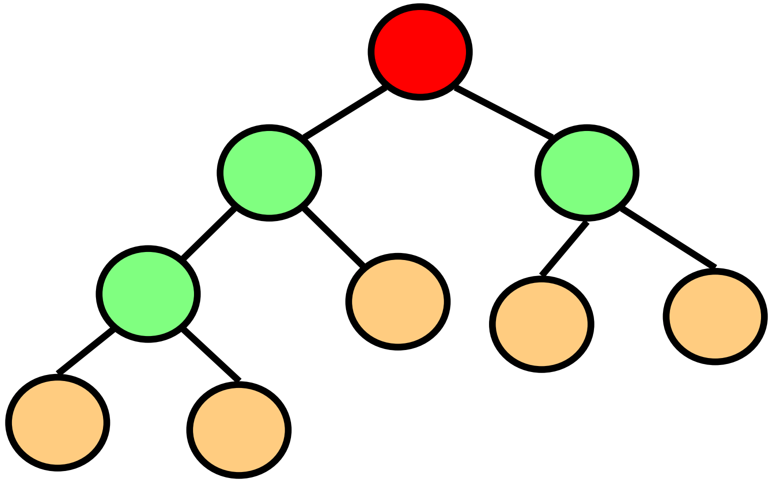


מהו הגובה המקסימלי והמינימלי
של עץ בינארי כתלות ב n ?

- יהי n מספר הצמתים בעץ בינארי כלשהו ויהי h גובה העץ

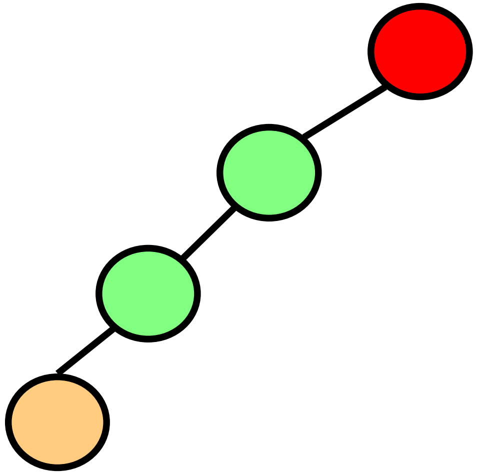
מינימלי : עץ (כמעט) שלם

$$h = \lfloor \log n \rfloor = \Theta(\log n)$$



מקסימלי : זיג-זג (שרשרת)
(שמאלי או ימני)

$$h = n - 1 = \Theta(n)$$



Traversals
סריקת עץ בינארי

I. **depth-first traversal – סריקה לעומק**

1. PreOrder traversal – סריקה תחילית
   - visit the parent first and then left and right children.
   - Root L R
2. InOrder traversal – סריקה תוכית
   - visit the left child, then the parent and the right child.
   - L Root R
3. PostOrder traversal – סריקה סופית
   - visit left child, then the right child and then the parent.
   - L R Root

II. **breadth-first traversal - סריקה לרוחב**

PreOrder Traversal
סריקה תחילית

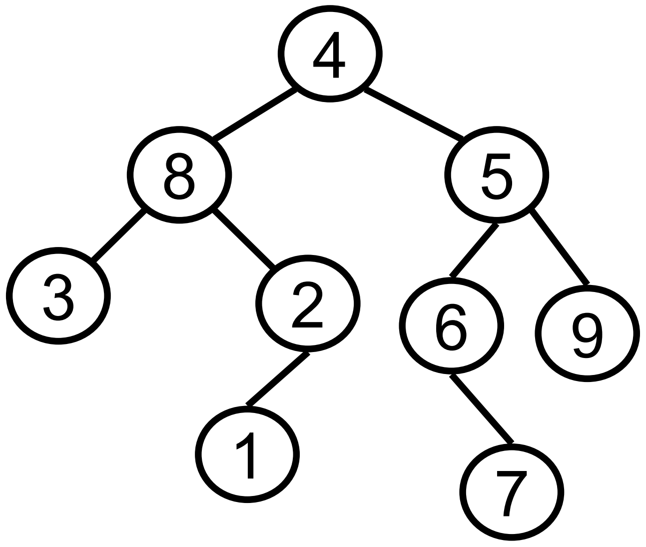
pseudo code

```
preOrder(root) {
  if root == null
     return
  print ( root )
  preOrder ( root.left )
  preOrder ( root.right )
}
```

Root L R

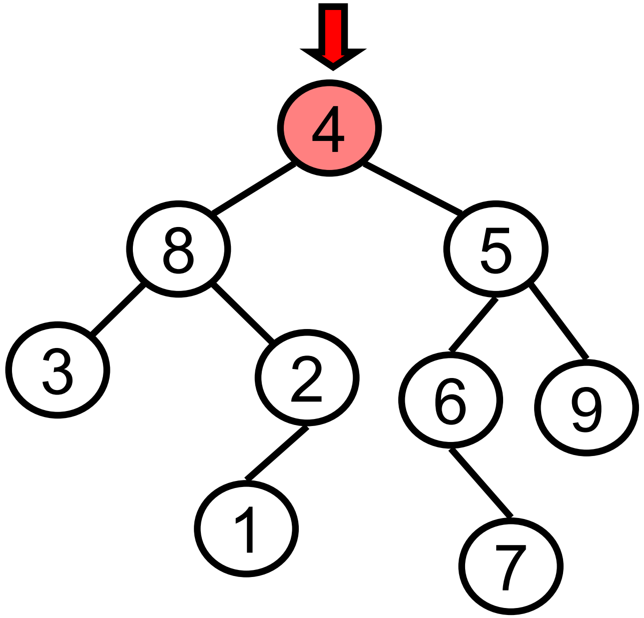
$\Theta(n)$

PreOrder Traversal
סריקה תחילית

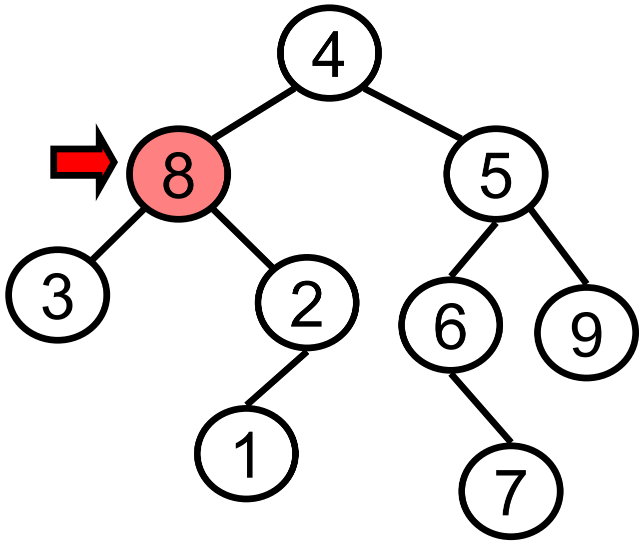


Root ➔ L➔ R : 4 8 3 2 1 5 6 7 9

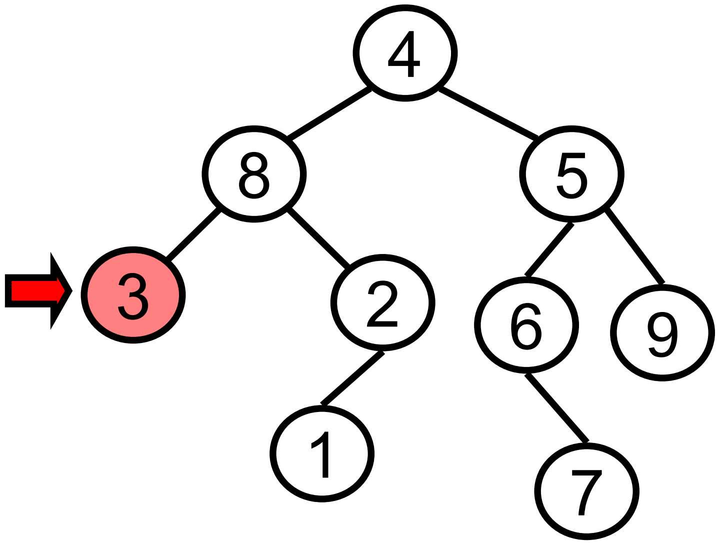
מעקב



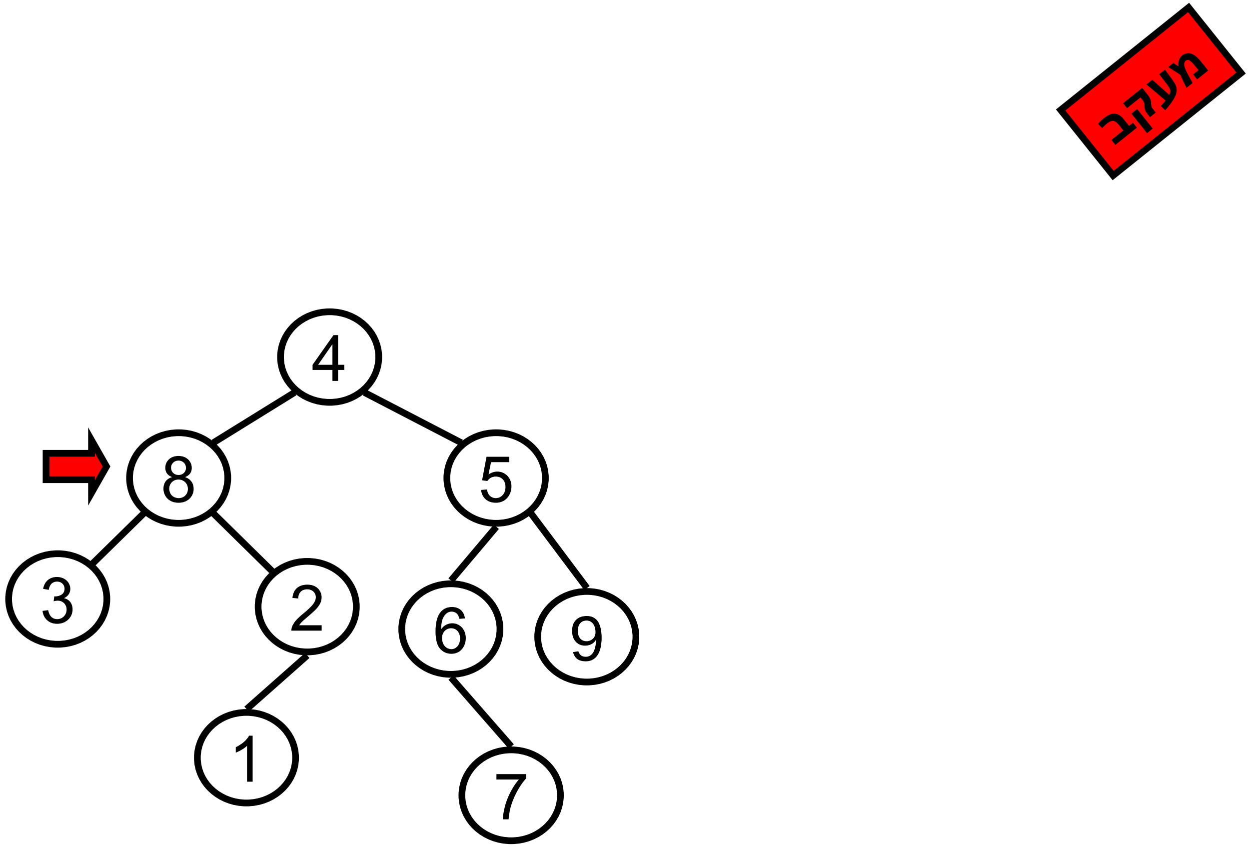
Root ➔ L➔ R : 4



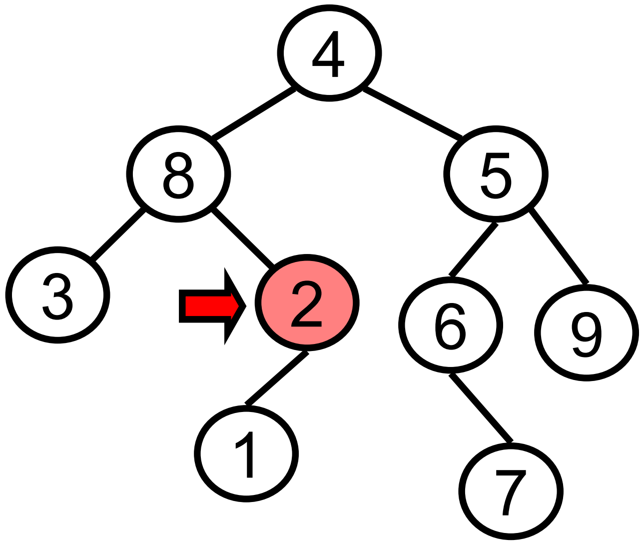
Root ➔ L➔ R : 4 8



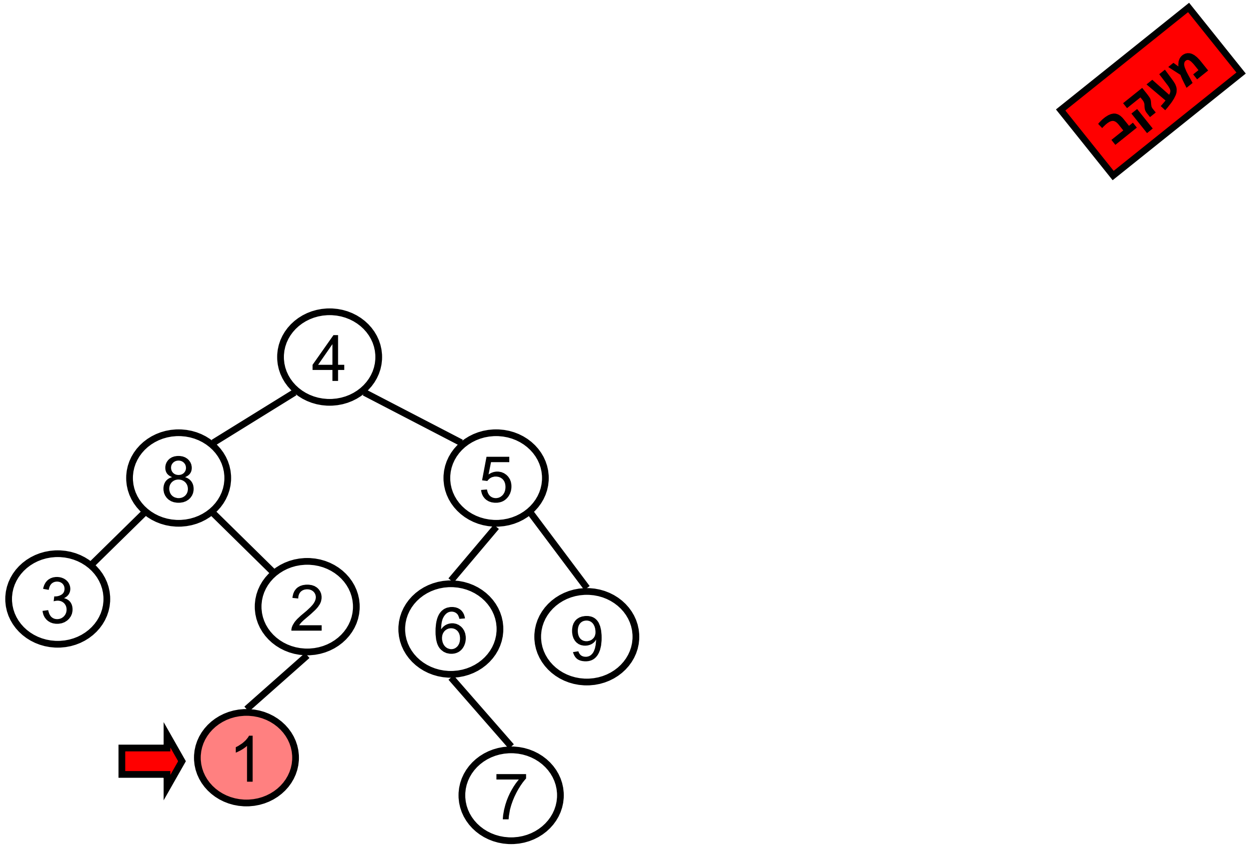
Root → L→ R : 4 8 3



Root ➔ L➔ R : 4 8 3

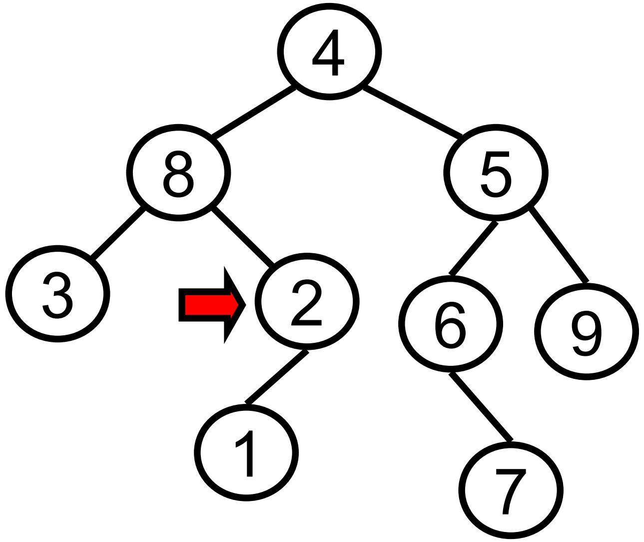


Root ➔ L➔ R : 4 8 3 2

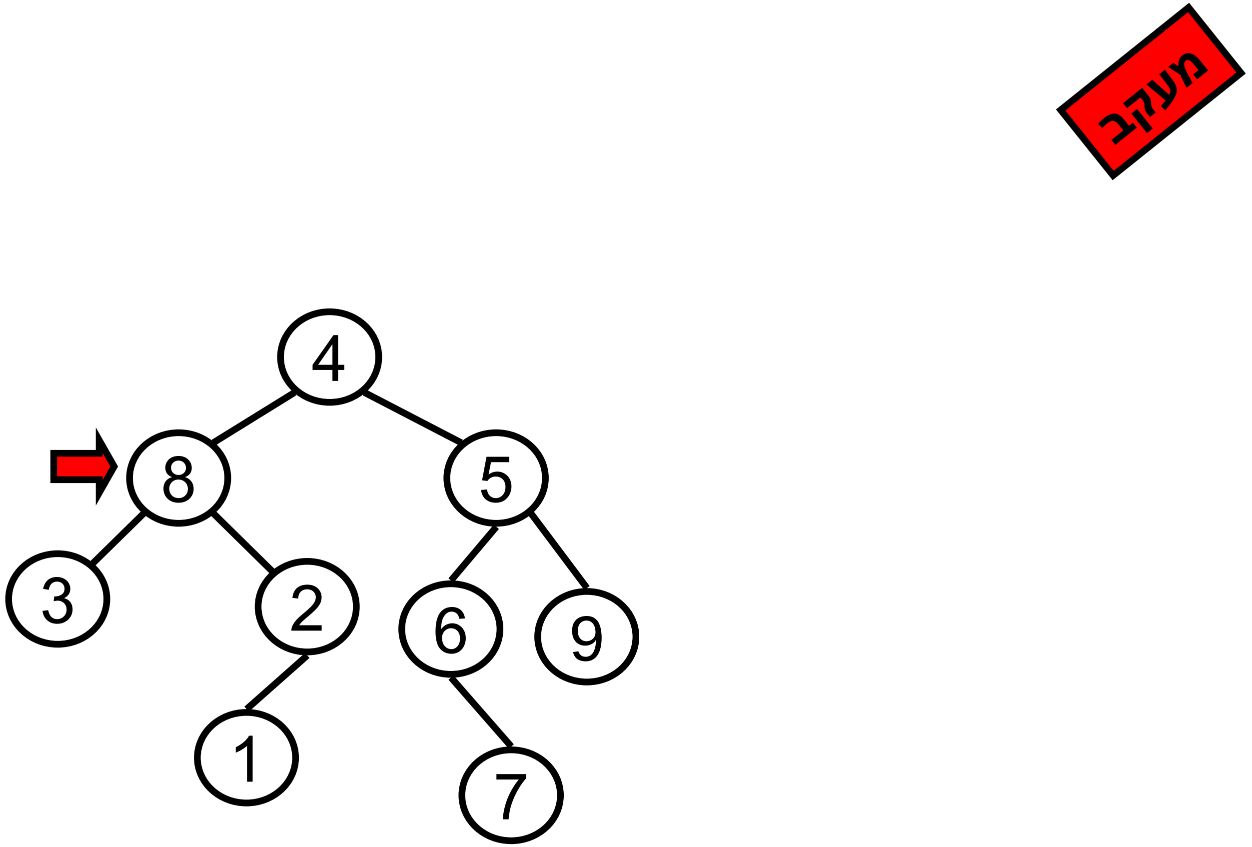


Root ➔ L➔ R : 4 8 3 2 1

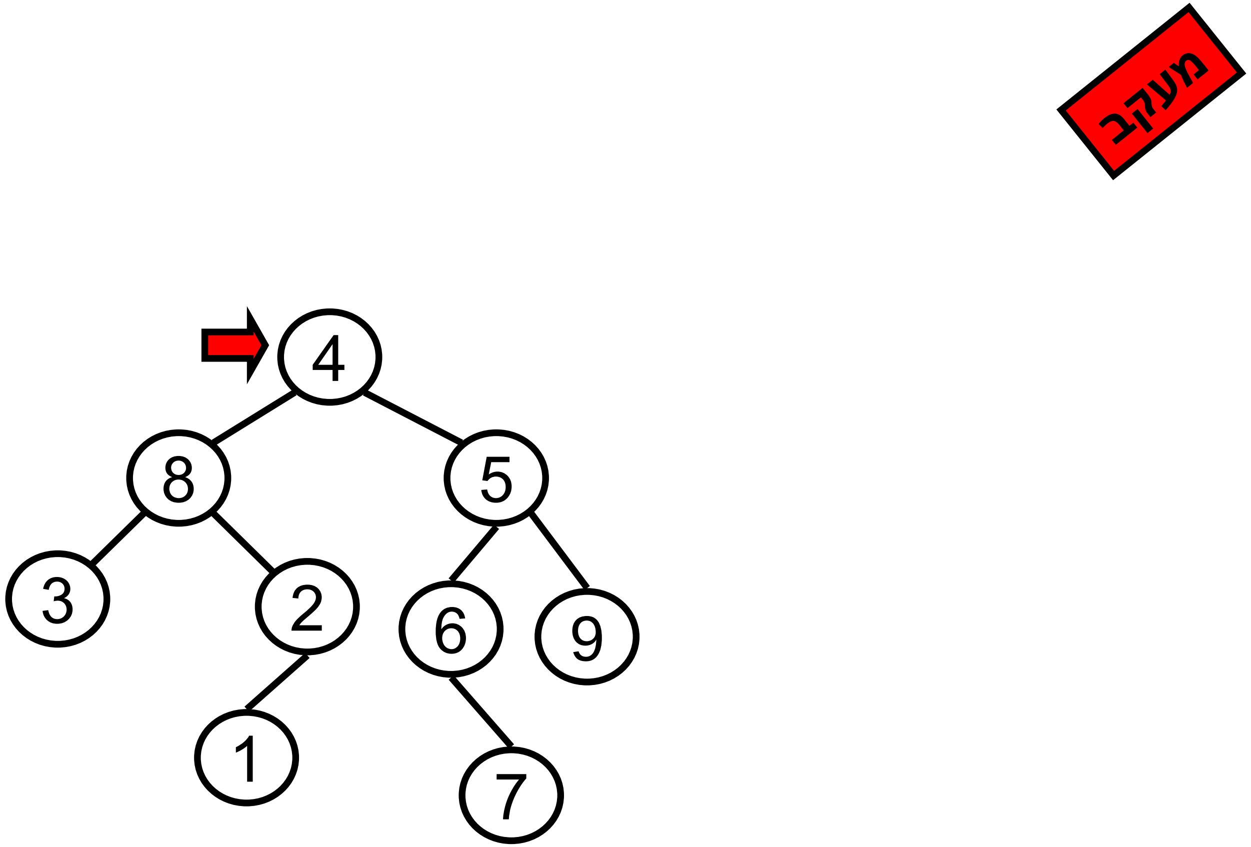
מעקב



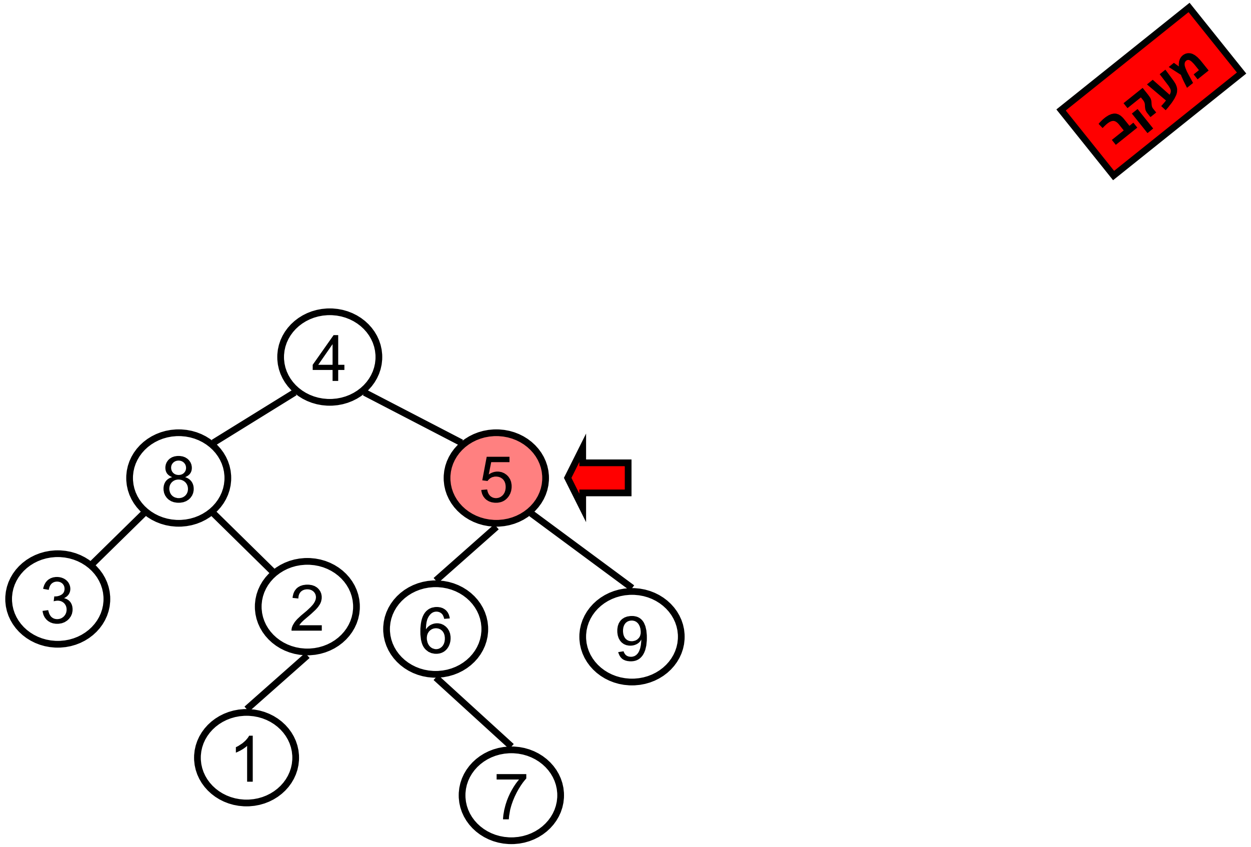
Root ➔ L➔ R : 4 8 3 2 1



Root ➔ L➔ R : 4 8 3 2 1

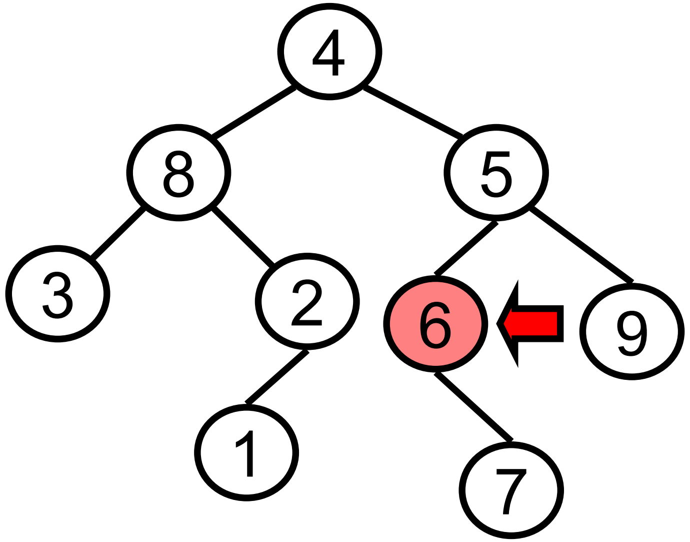


Root ➔ L➔ R : 4 8 3 2 1



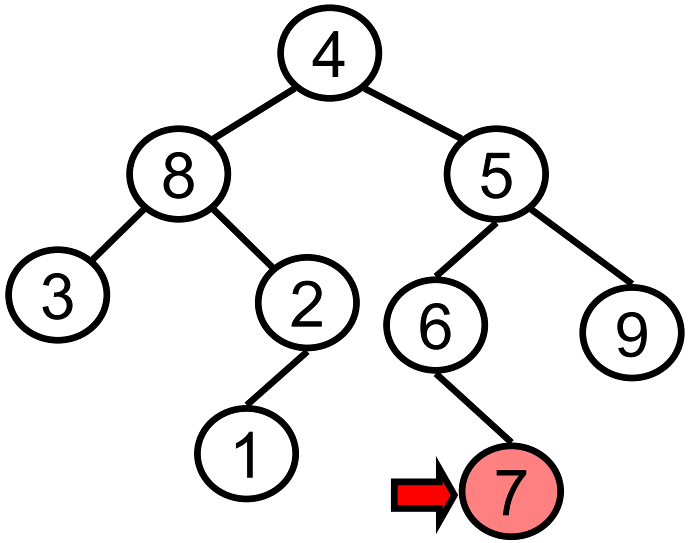
Root ➔ L➔ R : 4 8 3 2 1 5

מעקב

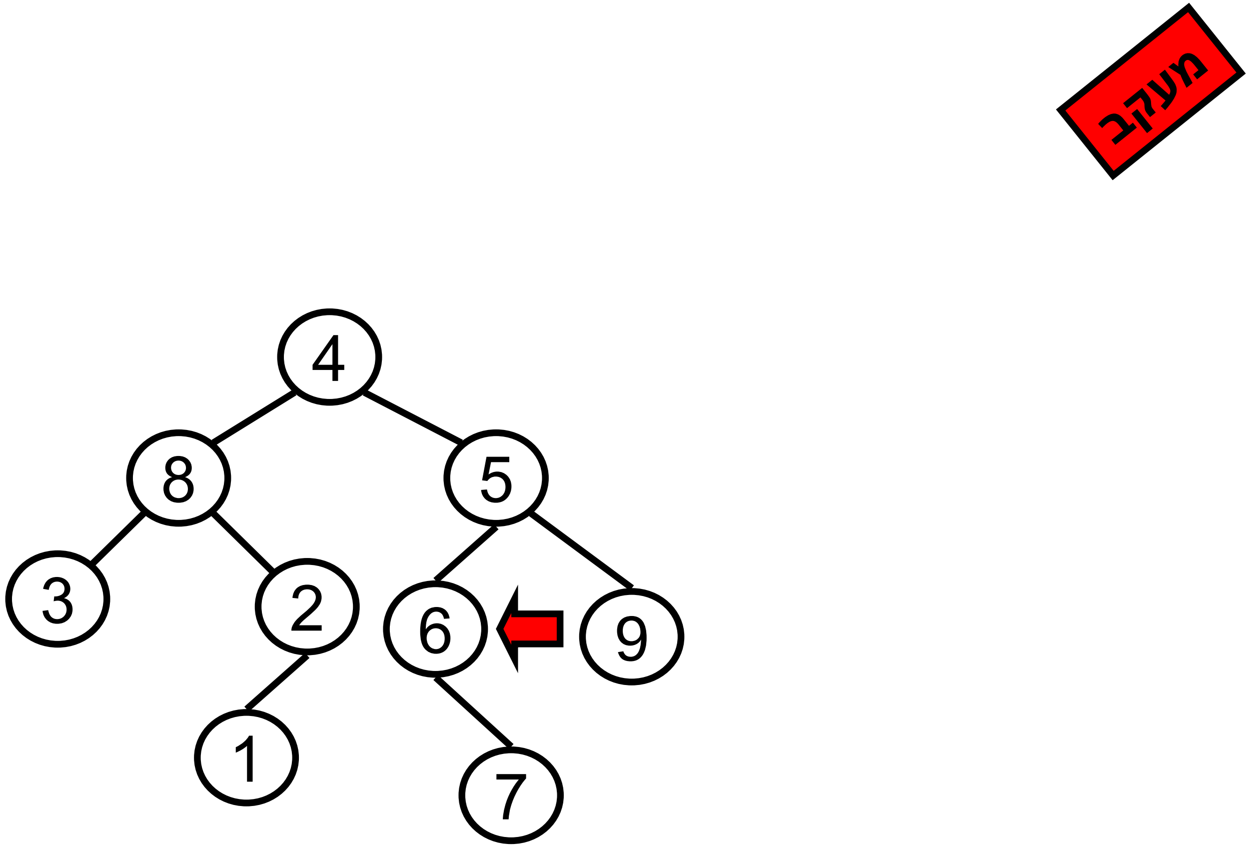


Root ➔ L➔ R : 4 8 3 2 1 5 6

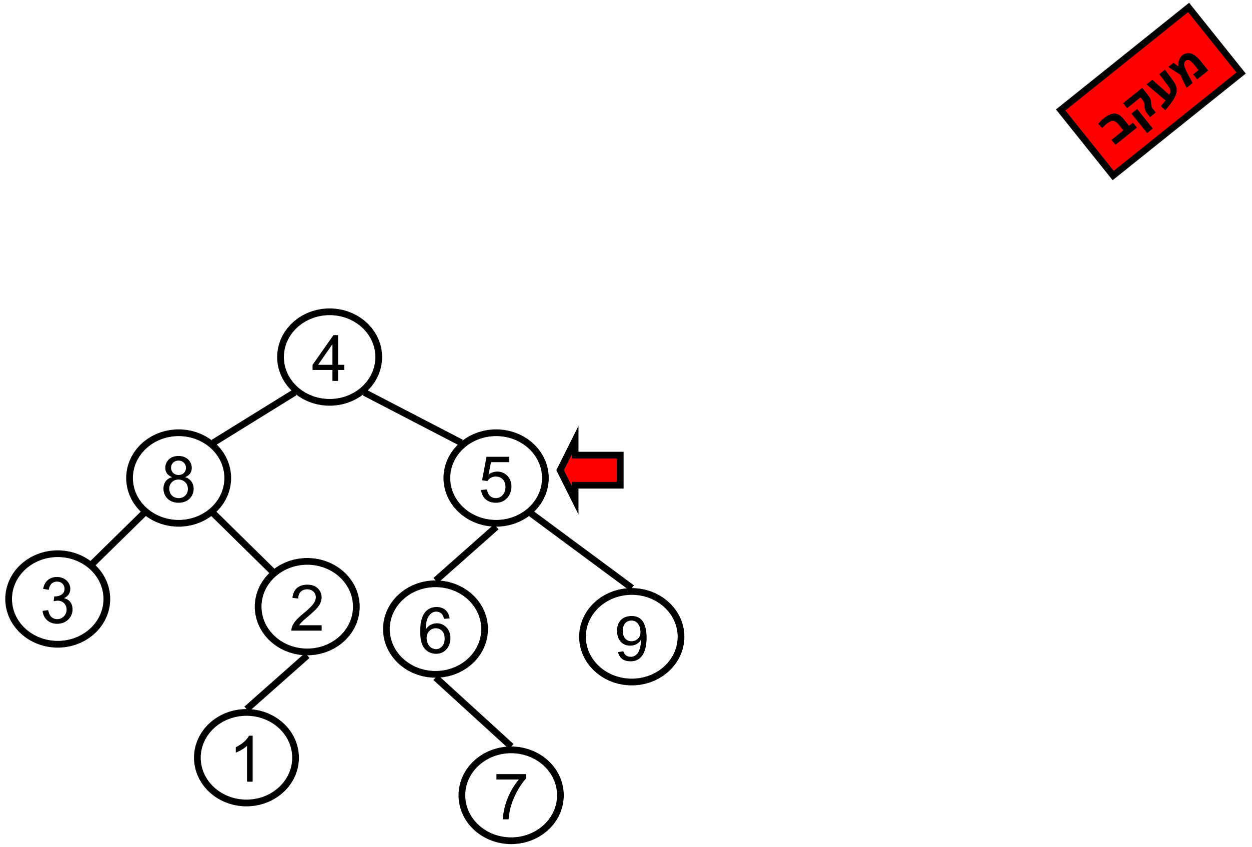
מעקב



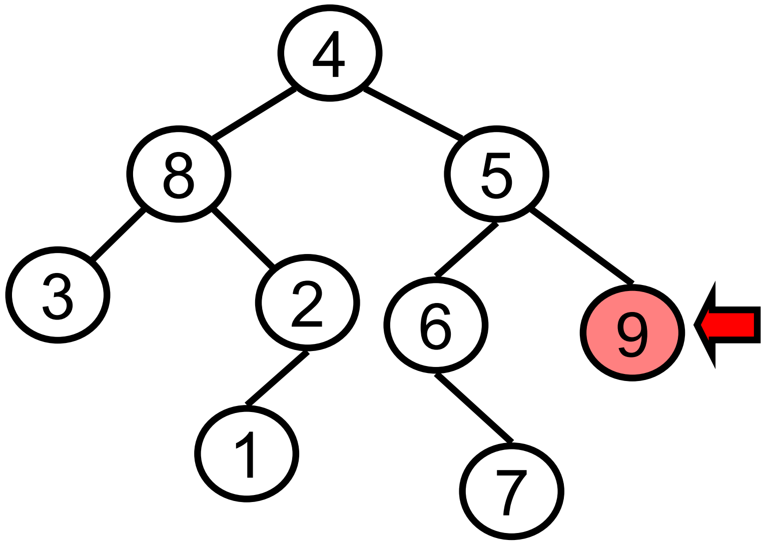
Root → L → R : 4 8 3 2 1 5 6 7



Root ➔ L➔ R : 4 8 3 2 1 5 6 7

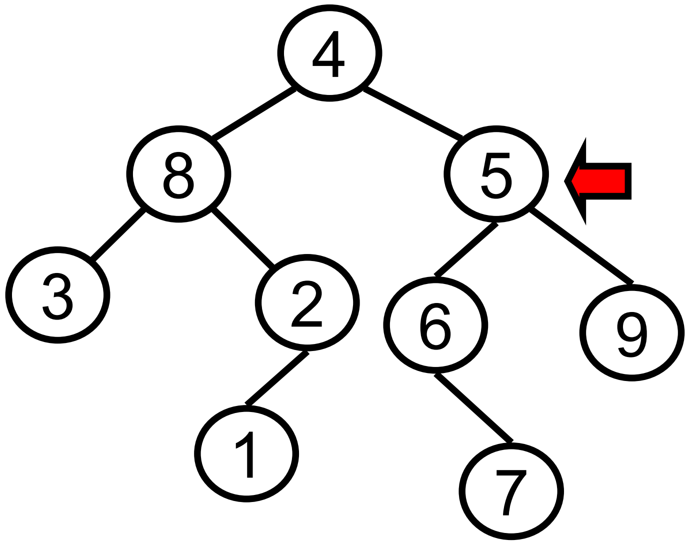


Root ➔ L➔ R : 4 8 3 2 1 5 6 7

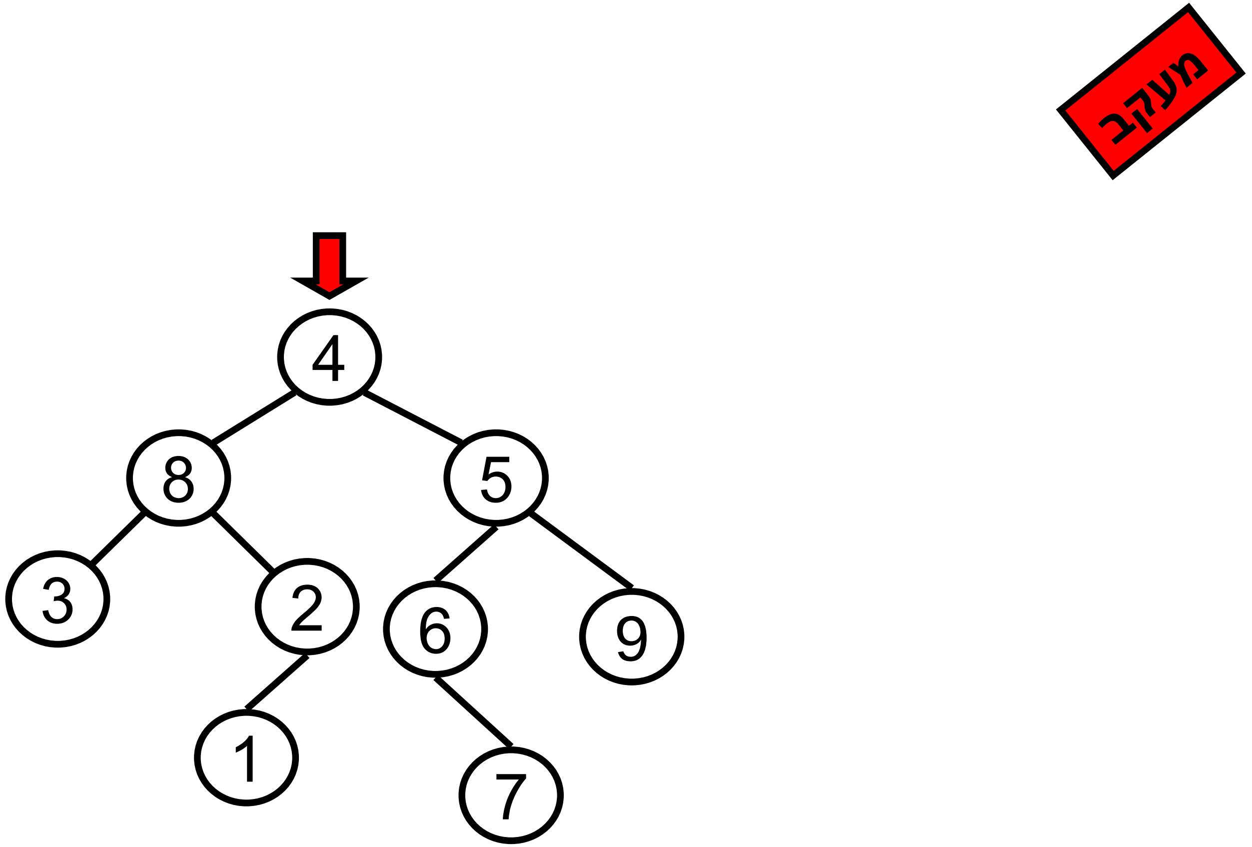


Root ➔ L➔ R : 4 8 3 2 1 5 6 7 9

מעקב



Root ➔ L➔ R : 4 8 3 2 1 5 6 7 9



Root ➔ L➔ R : 4 8 3 2 1 5 6 7 9

InOrder traversal
סריקה תוכית

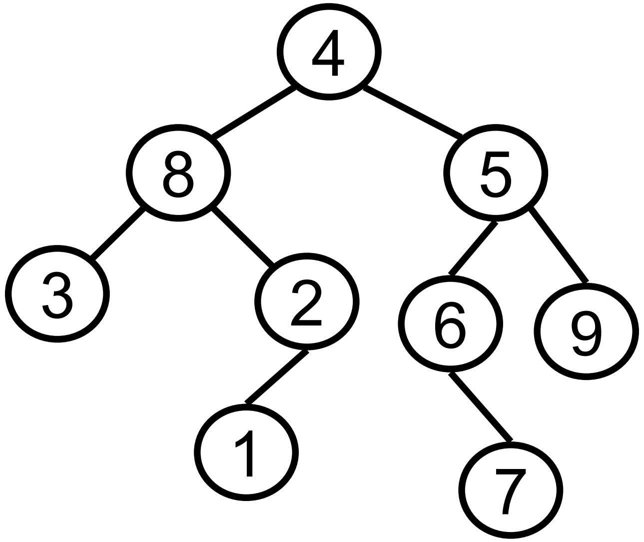
L Root R

Θ(n)

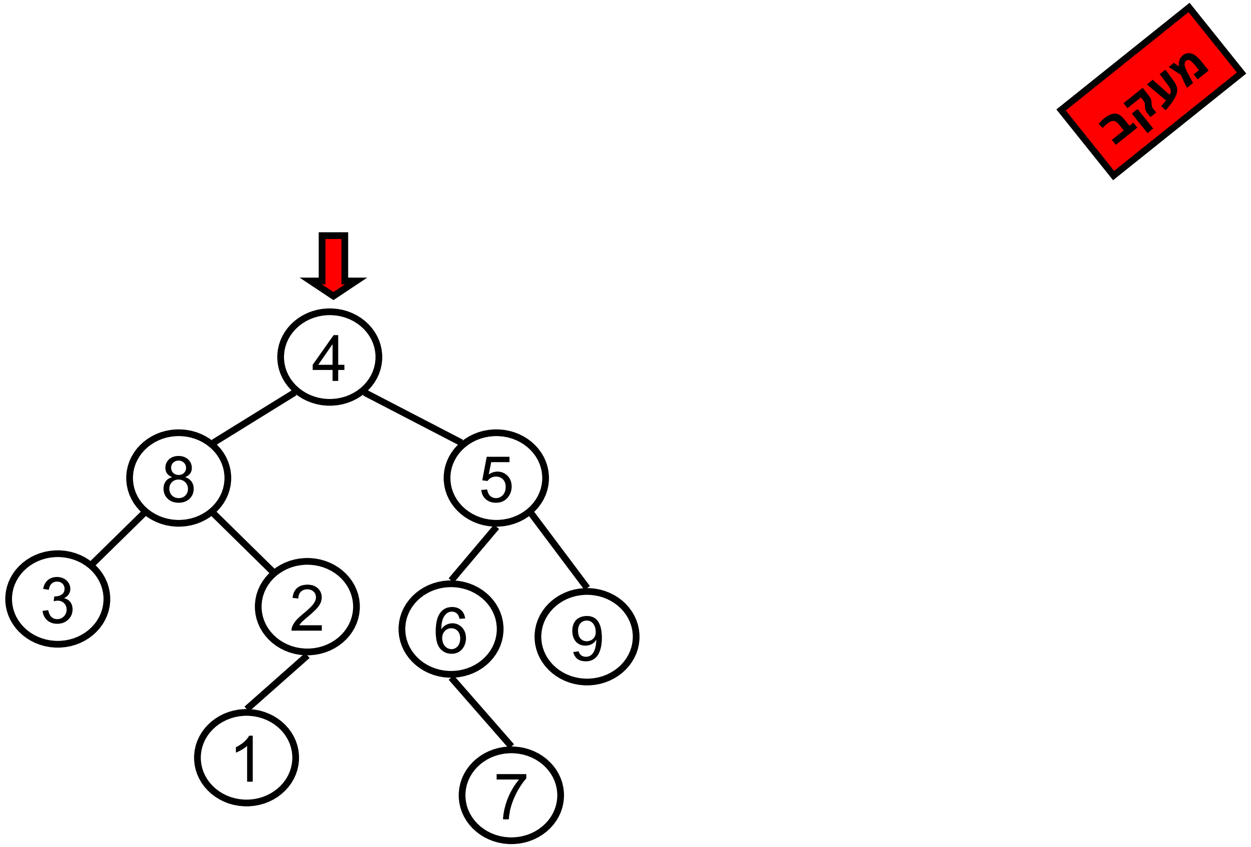
pseudo code

```
inOrder(root) {
  if root == null
     return
  inOrder ( root.left )
  print ( root )
  inOrder ( root.right )
}
```

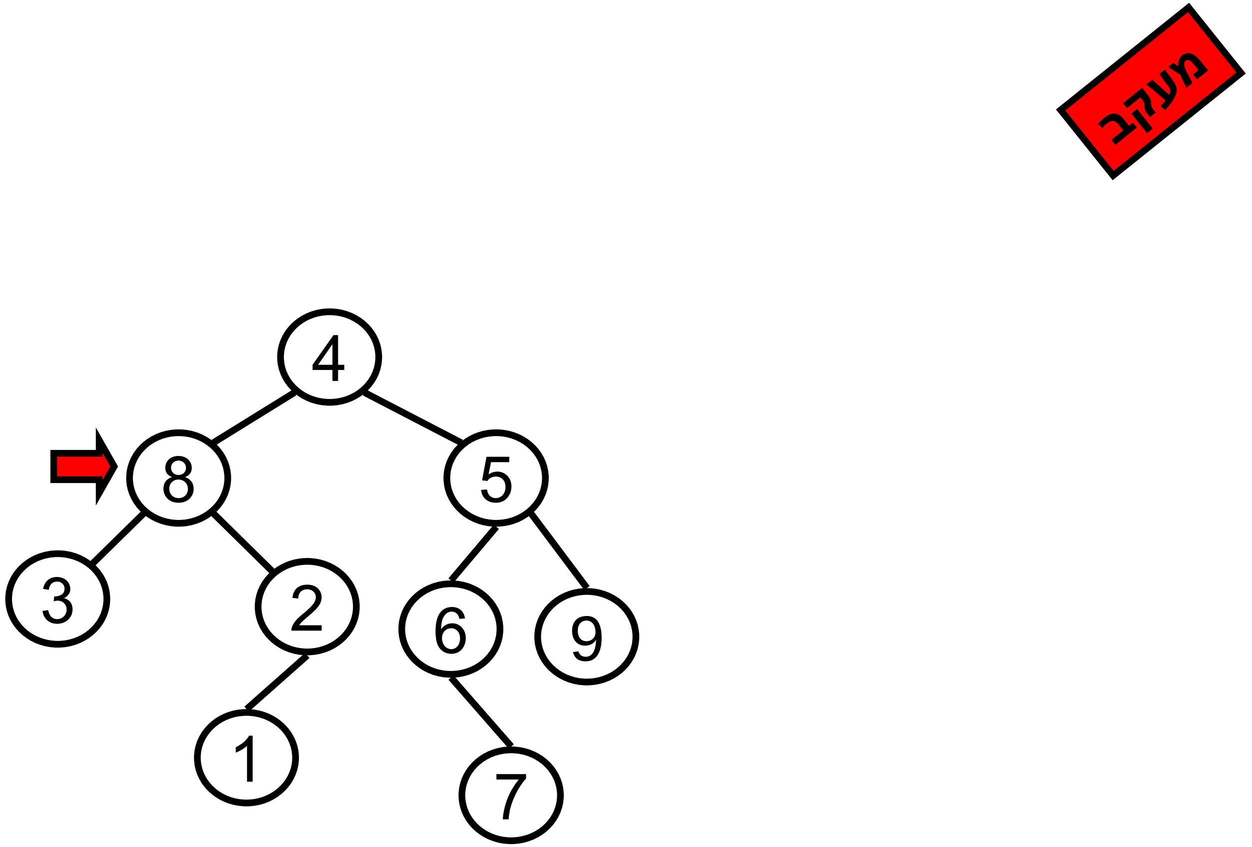
InOrder traversal
סריקה תוכית



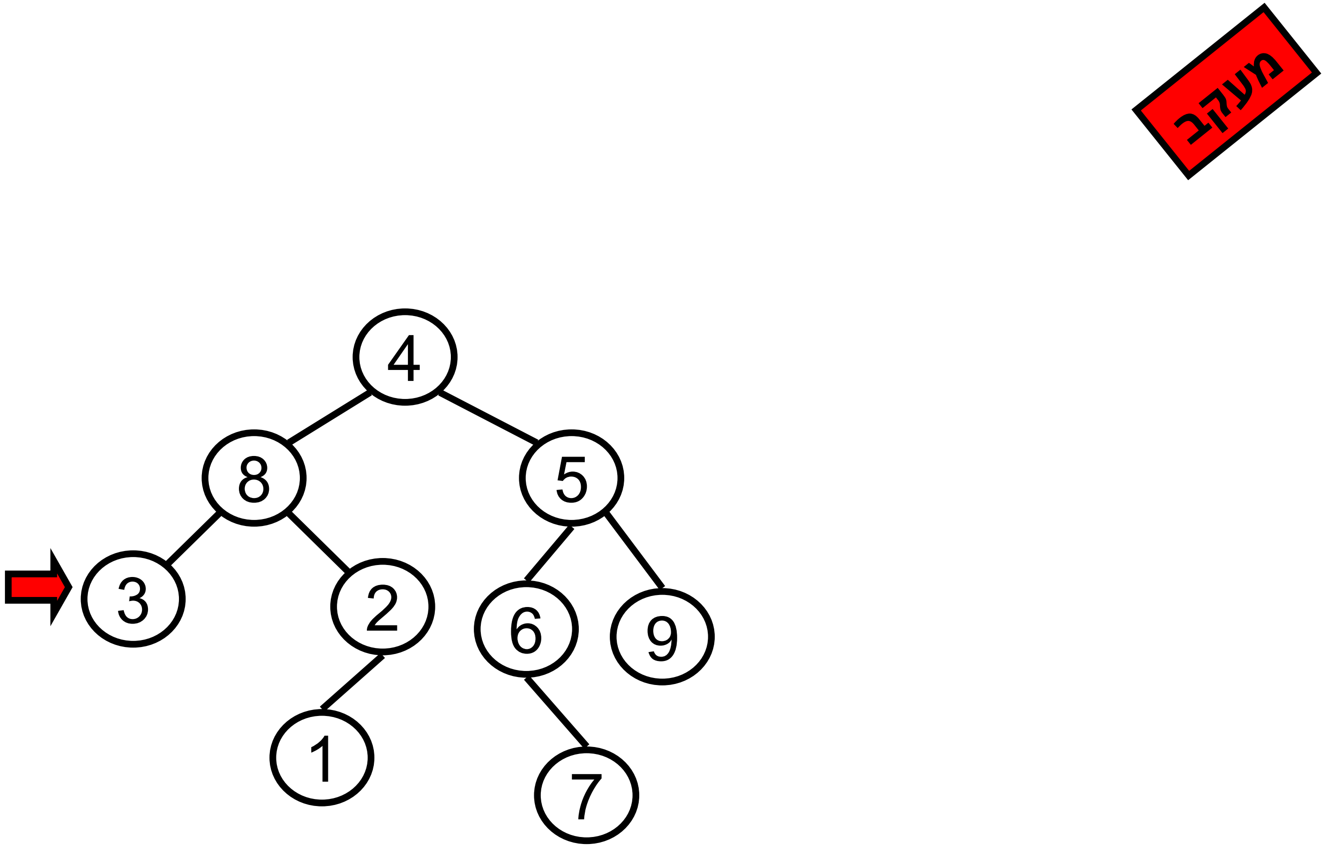
L ➔ Root ➔ R : 3 8 1 2 4 6 7 5 9



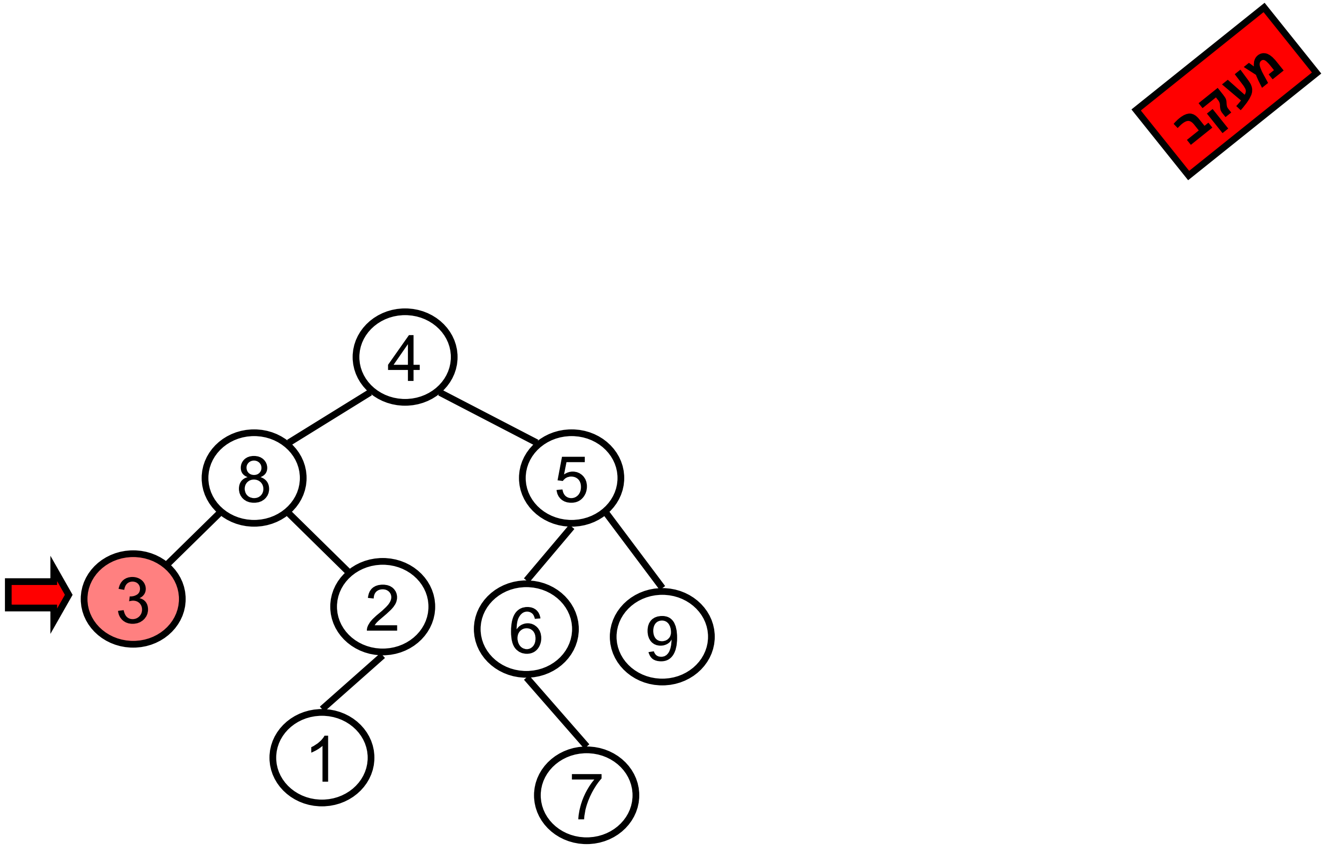
L ➔ Root ➔ R :



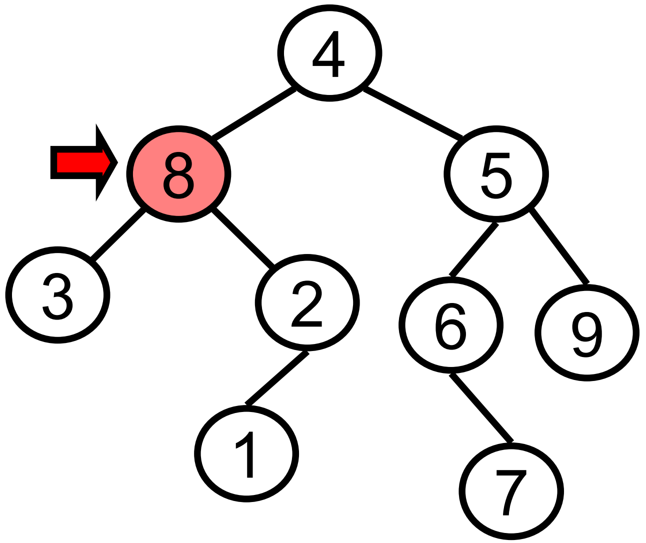
L ➔ Root ➔ R :



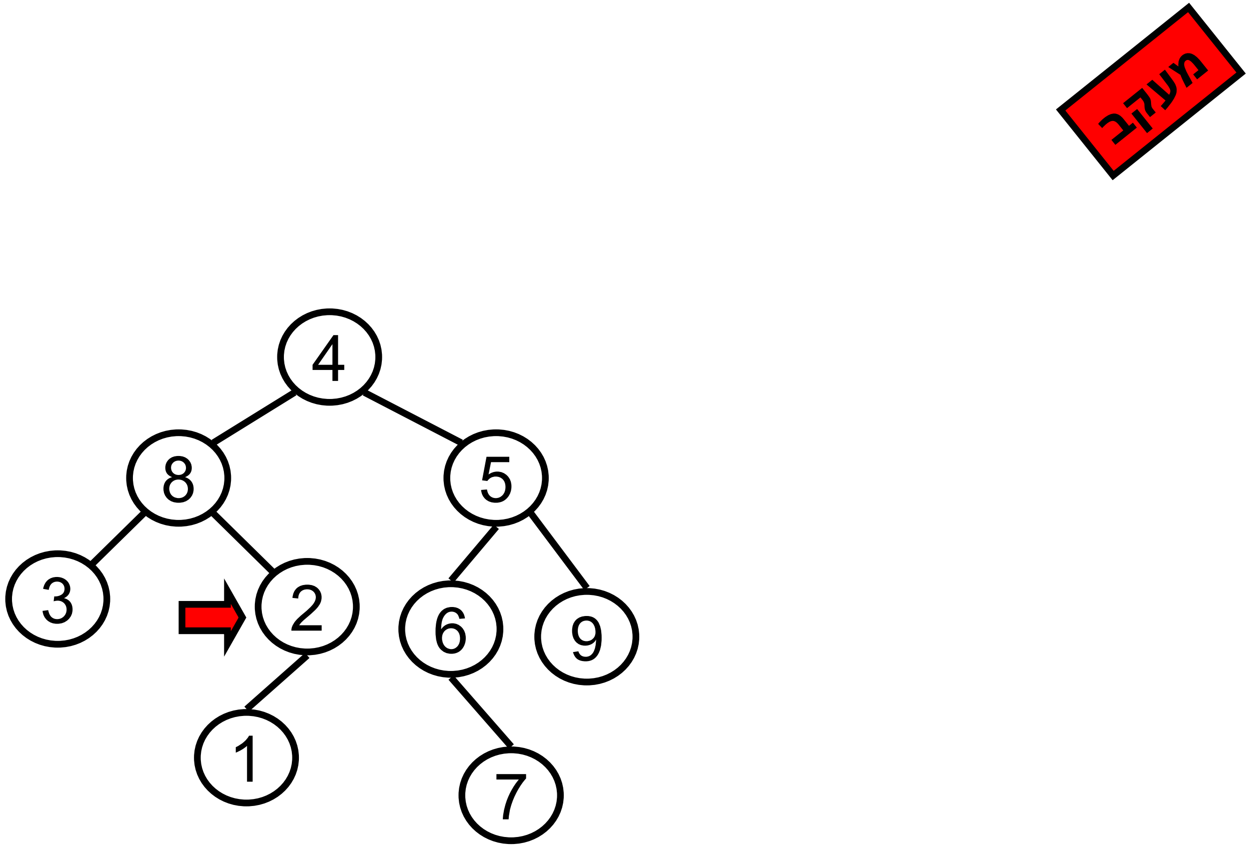
L ➔ Root ➔ R :



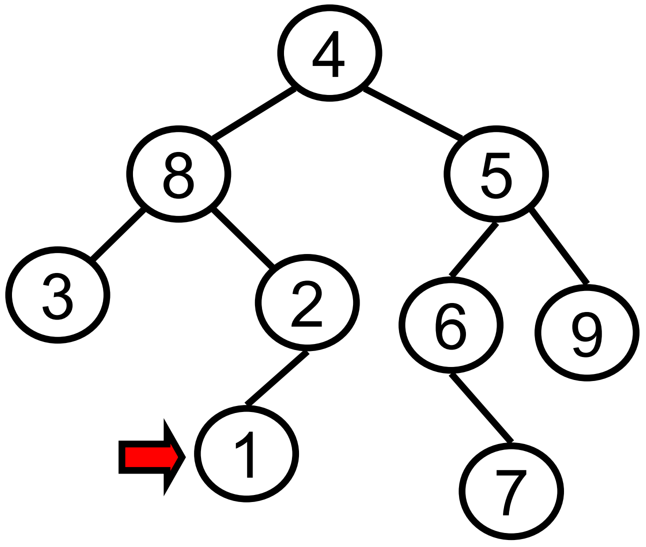
L ➔ Root ➔ R : 3



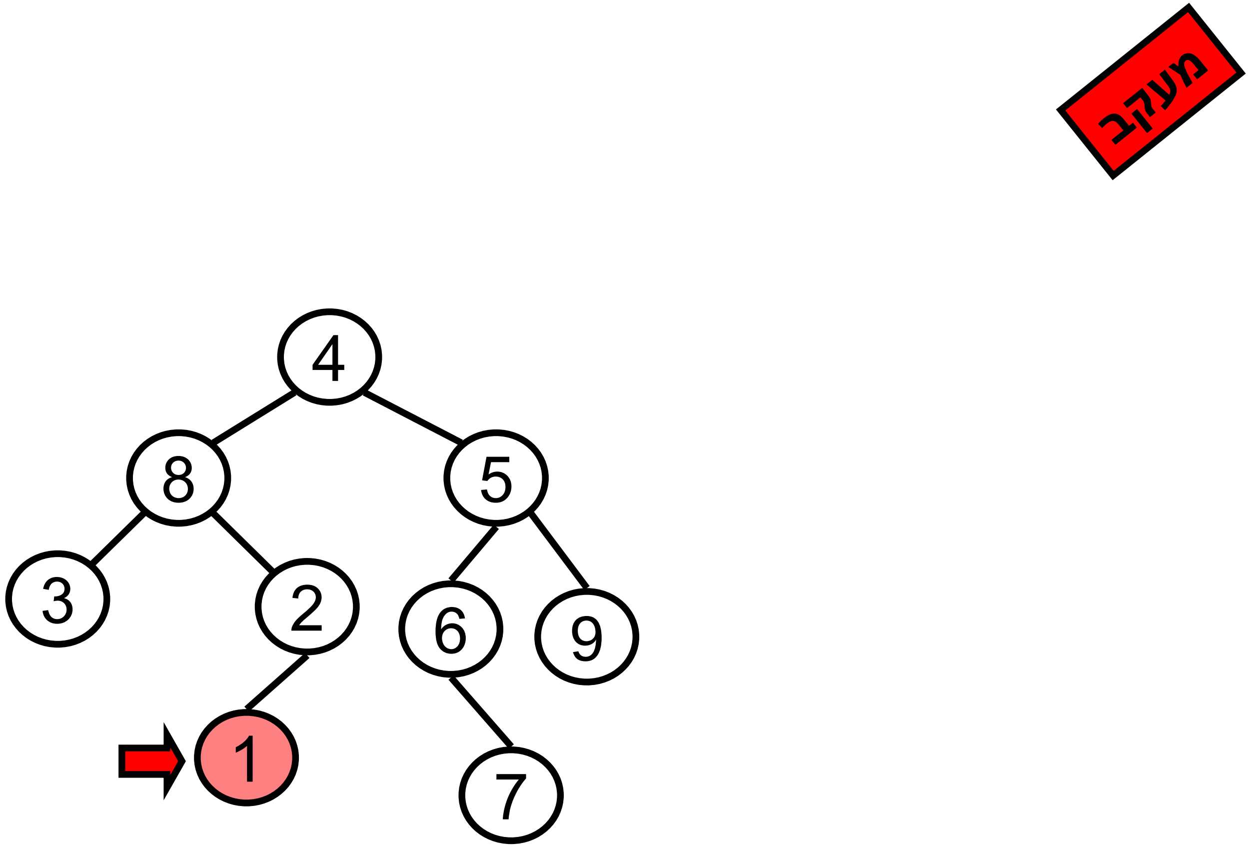
L ➔ Root ➔ R : 3 8



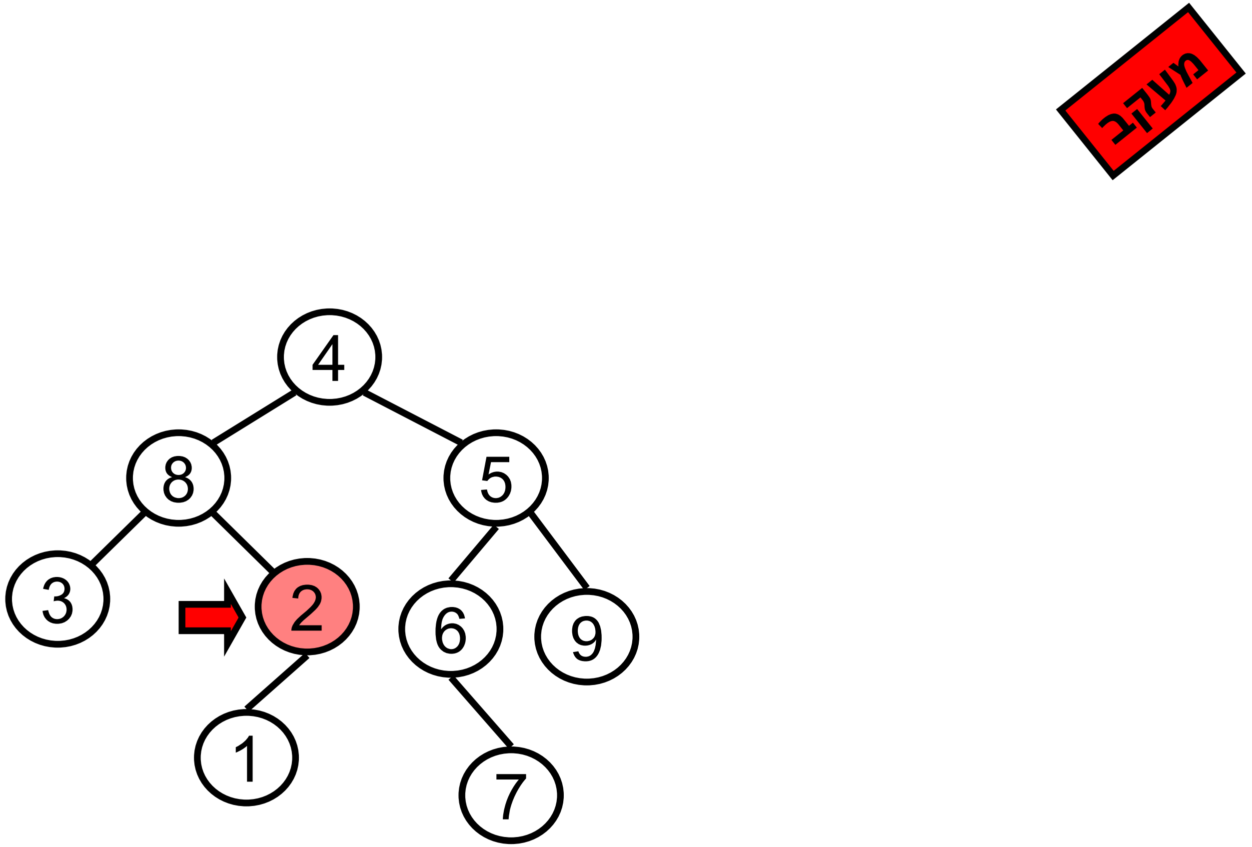
L ➔ Root ➔ R : 3 8



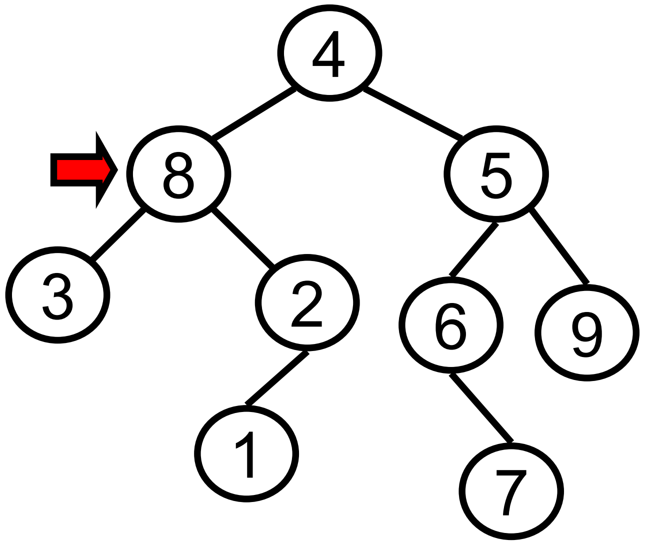
L → Root → R : 3 8



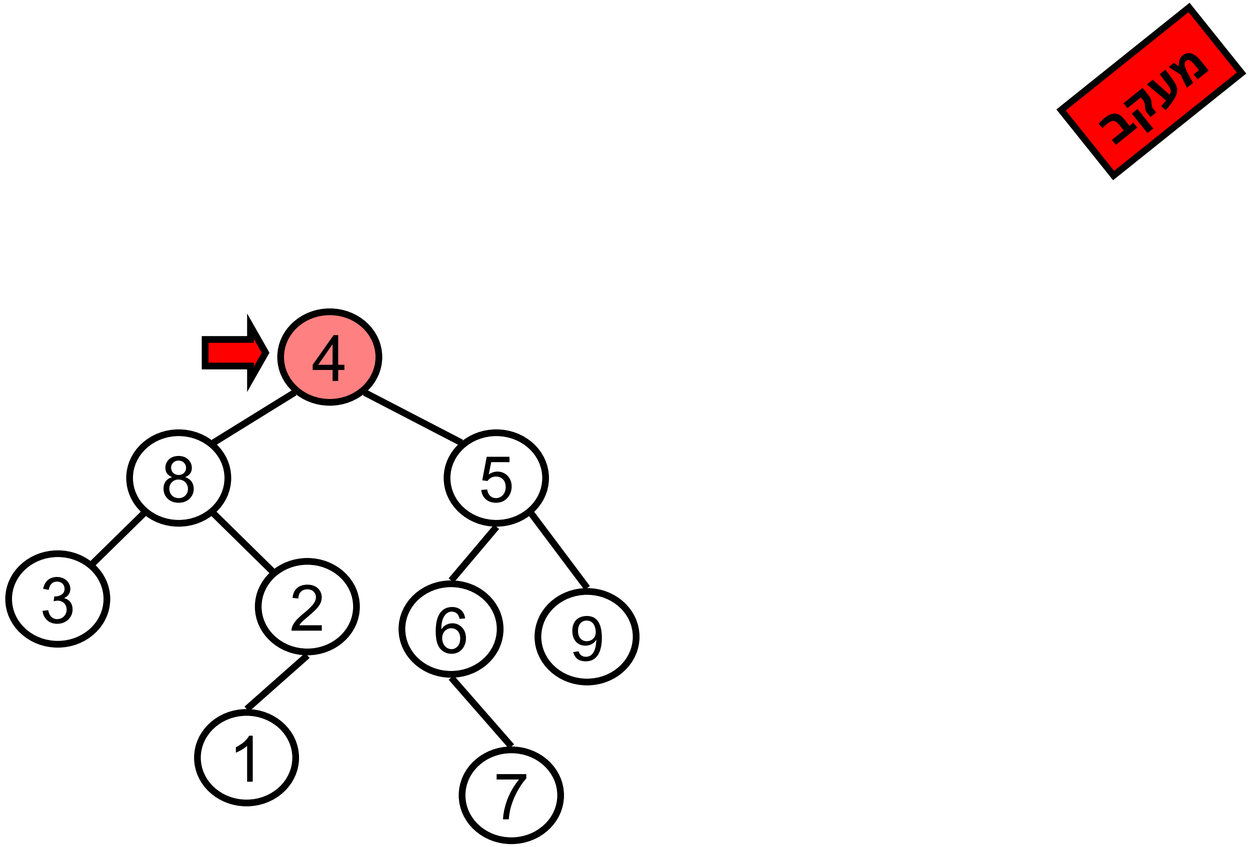
L ➔ Root ➔ R : 3 8 1



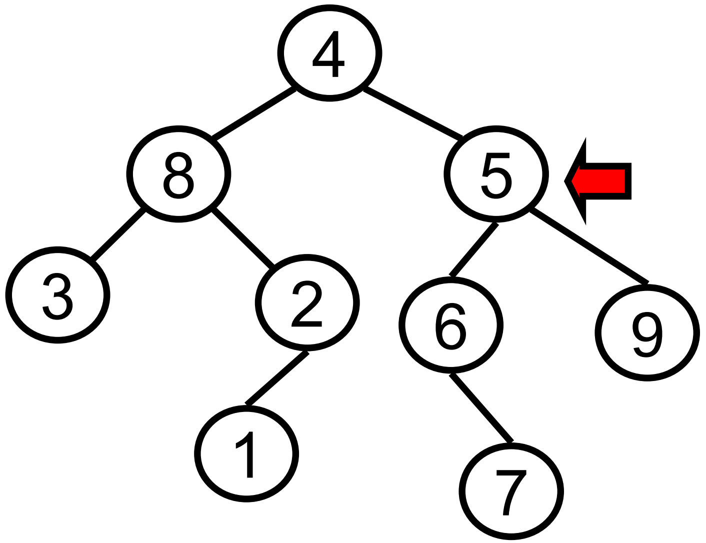
L ➔ Root ➔ R : 3 8 1 2



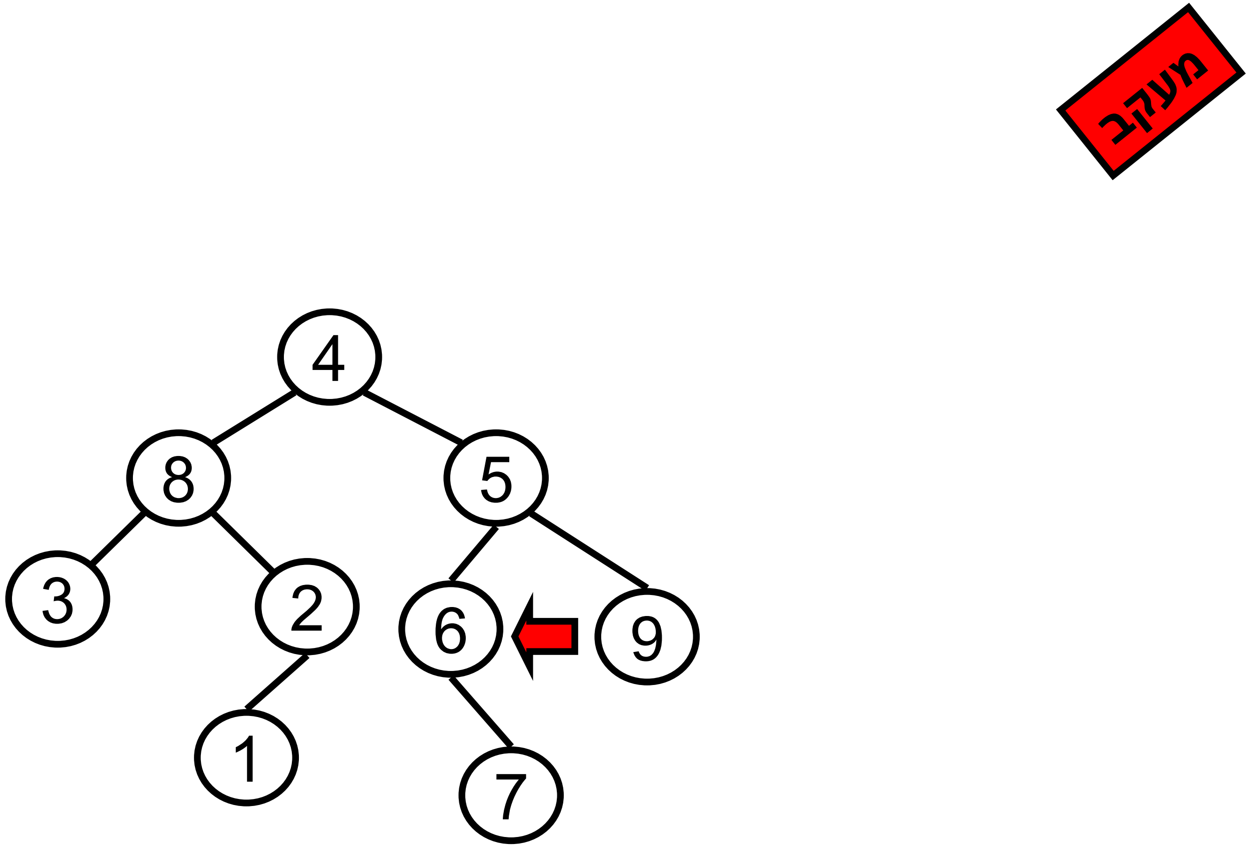
L → Root → R : 3 8 1 2



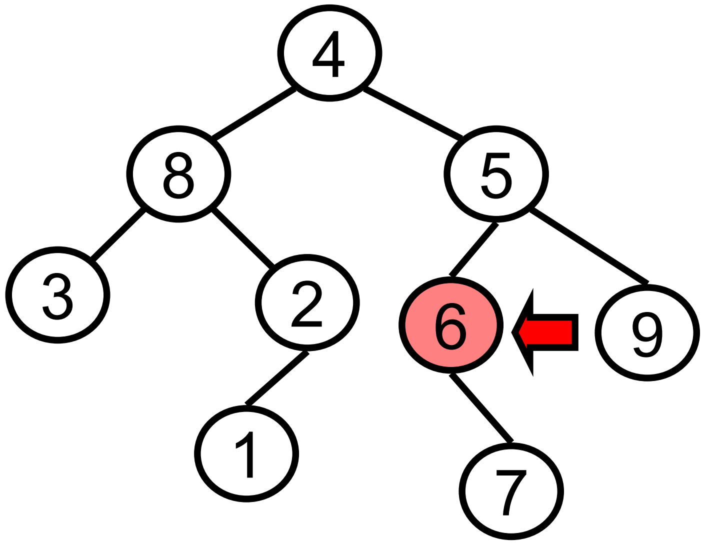
L → Root → R : 3 8 1 2 4



L → Root → R : 3 8 1 2 4

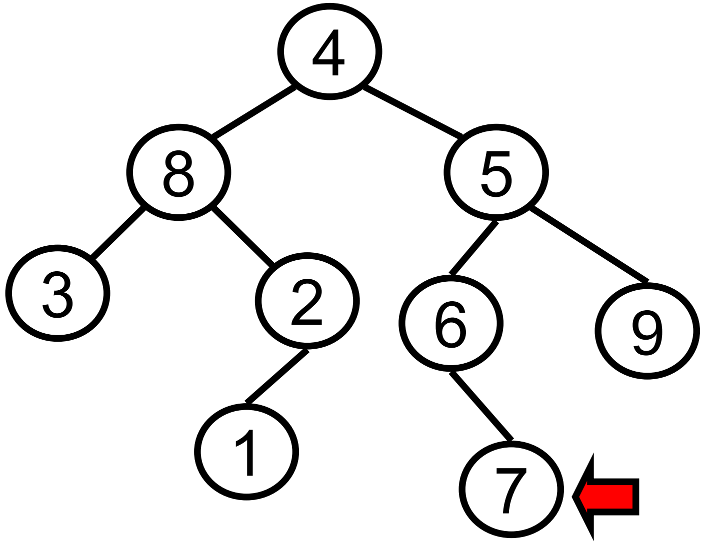


L → Root → R : 3 8 1 2 4

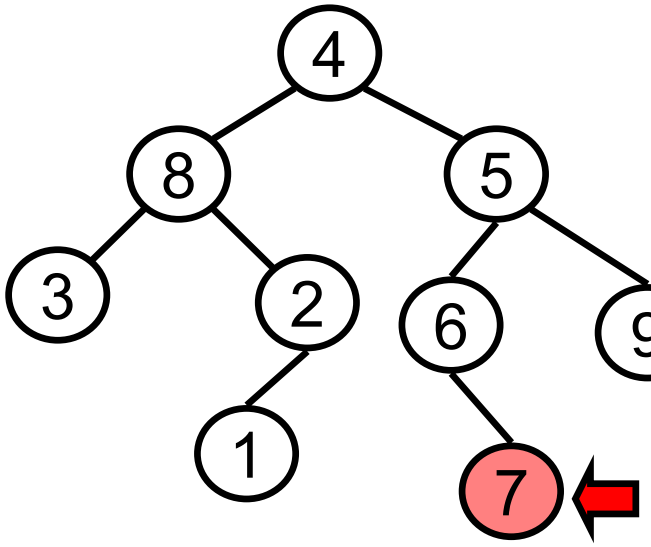


L ➔ Root ➔ R : 3 8 1 2 4 6

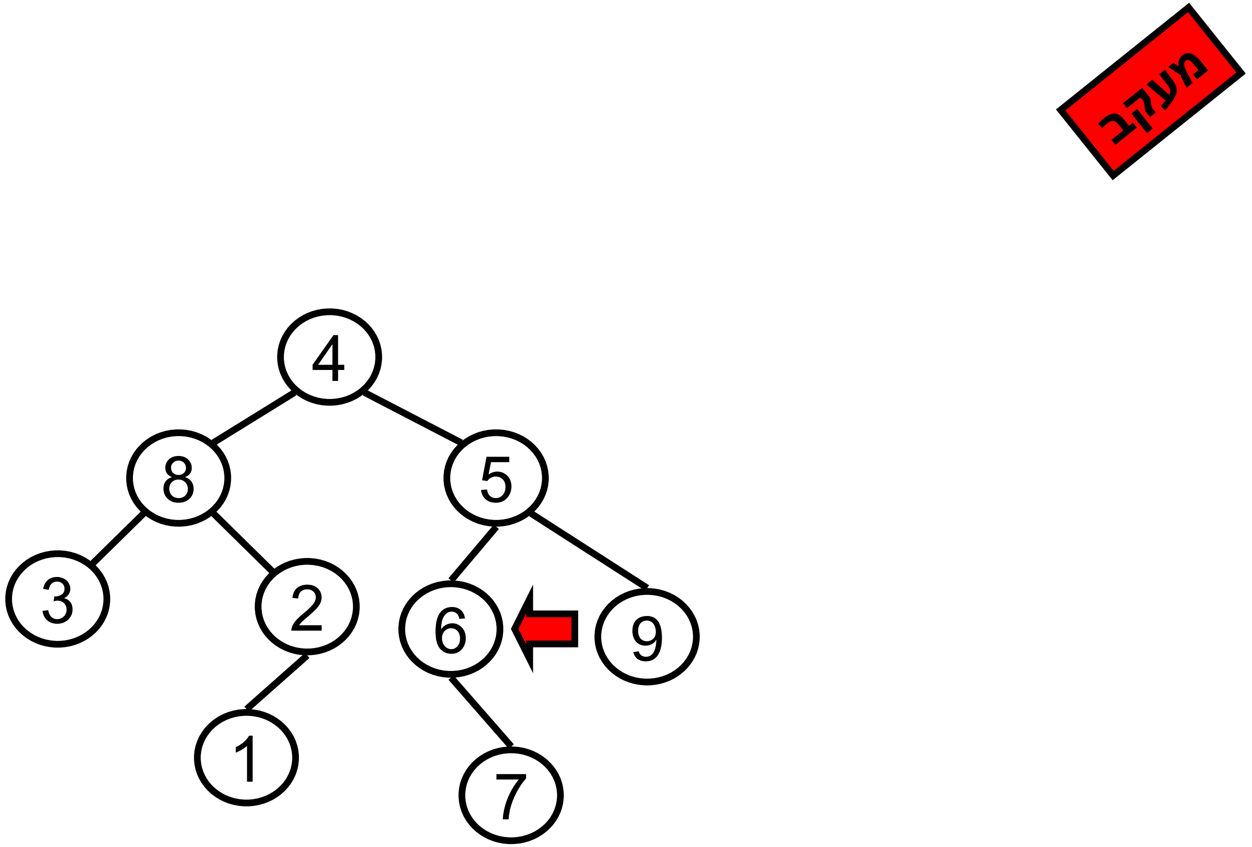
מעקב



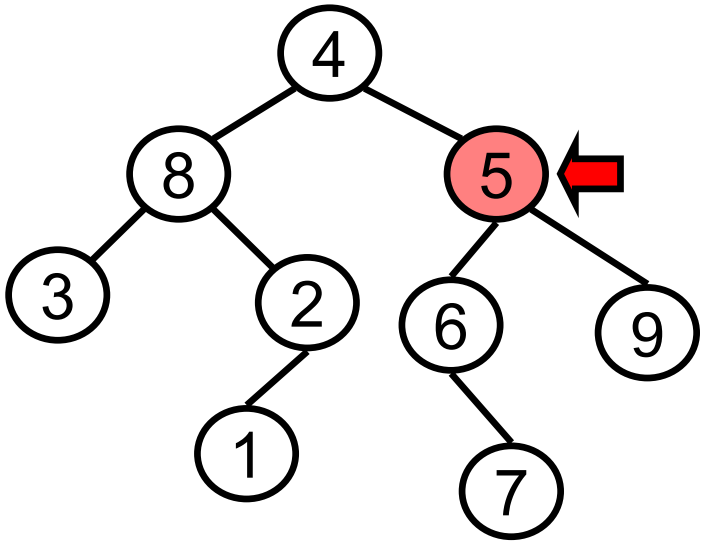
L ➔ Root ➔ R : 3 8 1 2 4 6



L → Root → R : 3 8 1 2 4 6 7

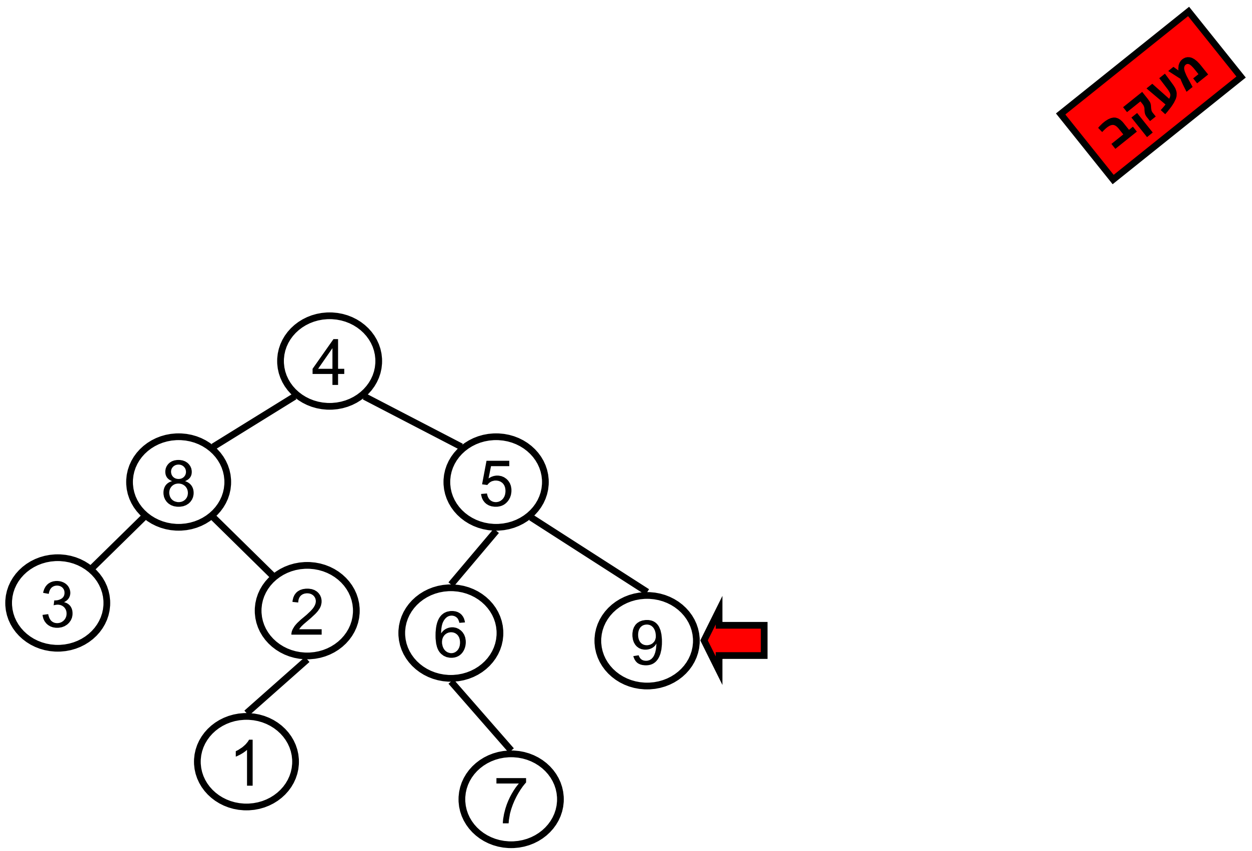


L ➔ Root ➔ R : 3 8 1 2 4 6 7

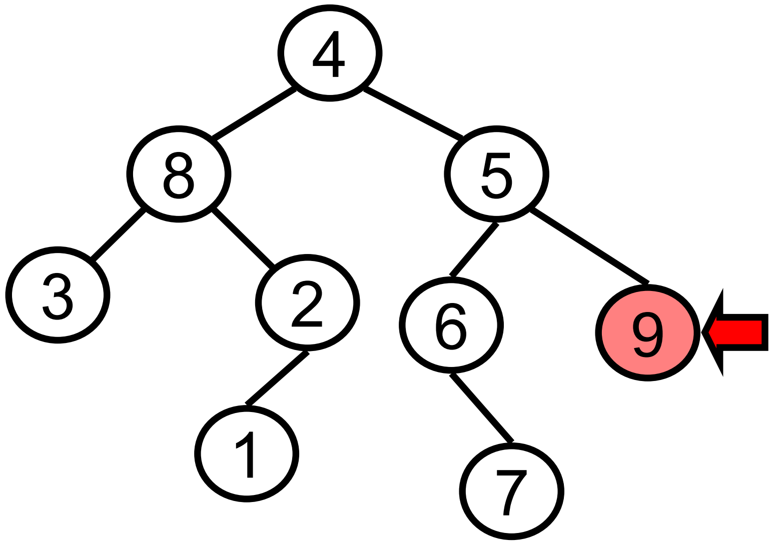


מעקב

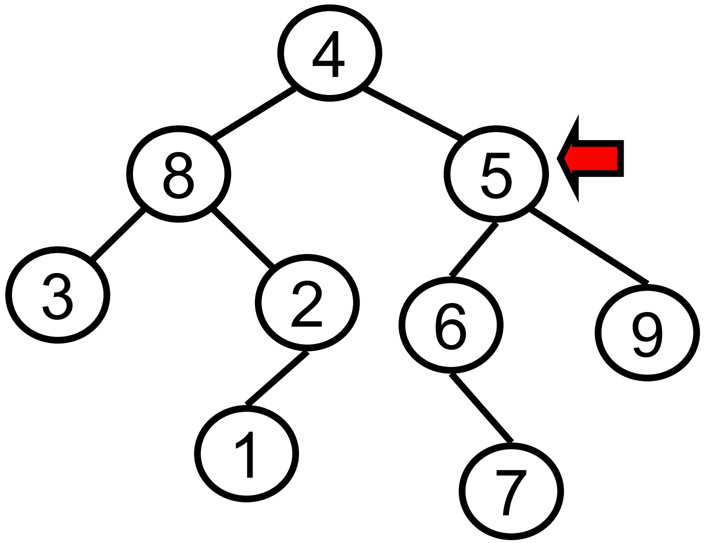
L ➔ Root ➔ R : 3 8 1 2 4 6 7 5



L → Root → R : 3 8 1 2 4 6 7 5

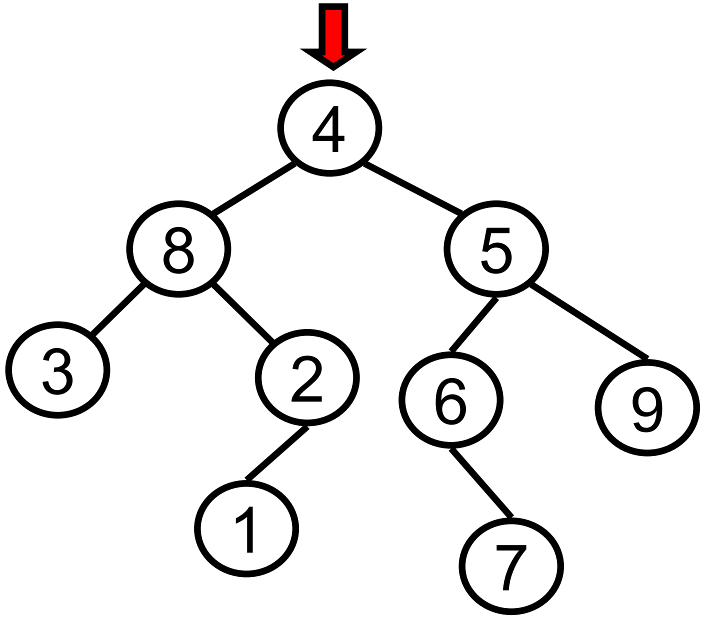


L → Root → R : 3 8 1 2 4 6 7 5 9



L → Root → R : 3 8 1 2 4 6 7 5 9

מעקב



L → Root → R : 3 8 1 2 4 6 7 5 9

PostOrder traversal
סריקה סופית

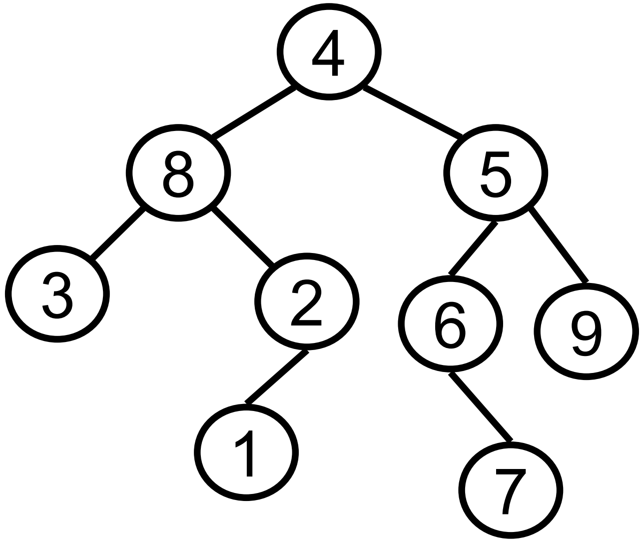
L R Root

pseudo code

```
postOrder(root) {
  if root == null
     return
  postOrder ( root.left )
  postOrder ( root.right )
  print ( root )
}
```

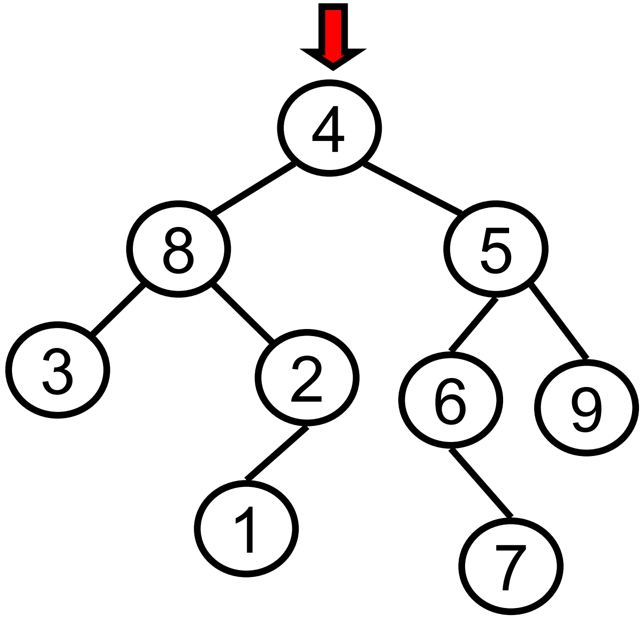
Θ(n)

PostOrder traversal
סריקה סופית

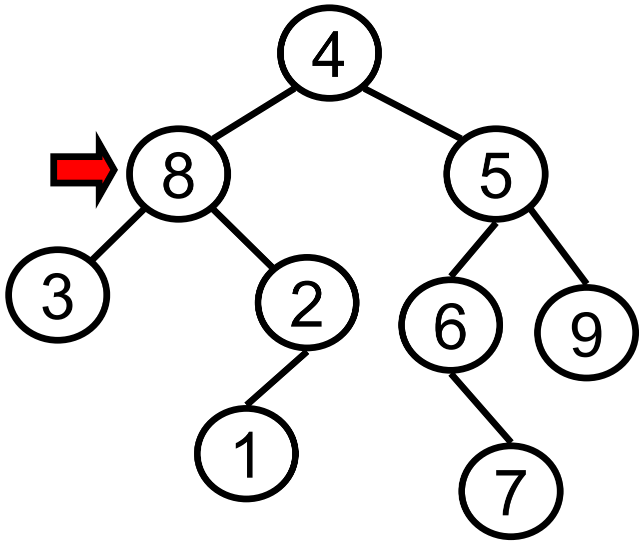


L ➔ R ➔ Root : 3 1 2 8 7 6 9 5 4

מעקב

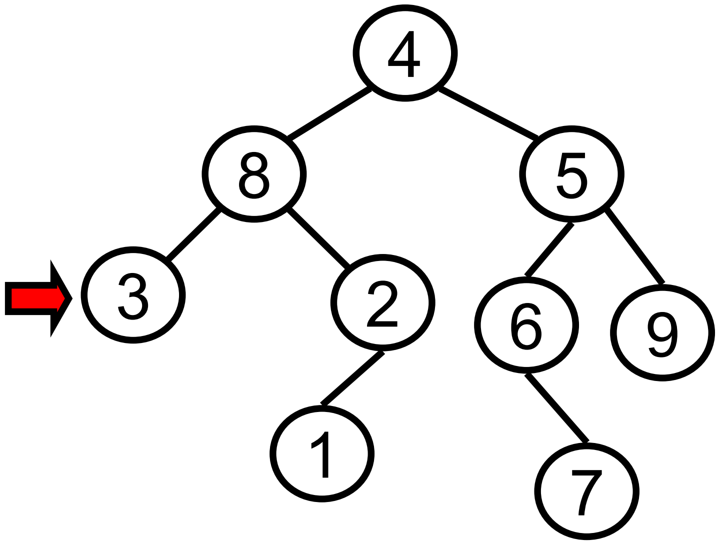


L ➔ R ➔ Root :



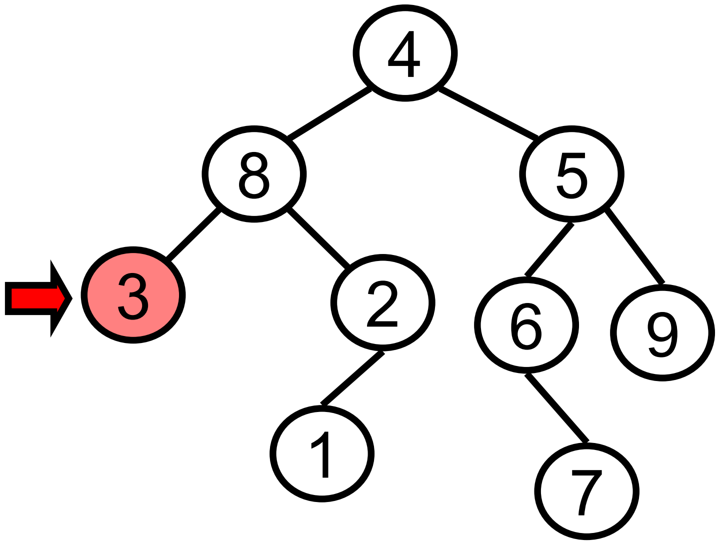
L ➔ R ➔ Root :

מעקב

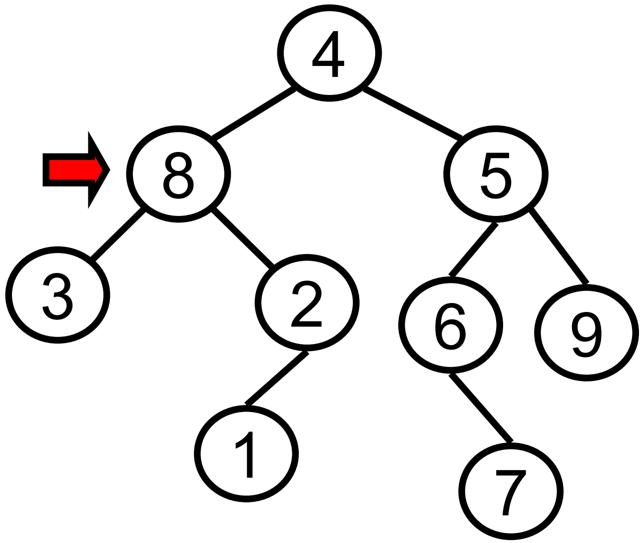


L ➔ R ➔ Root :

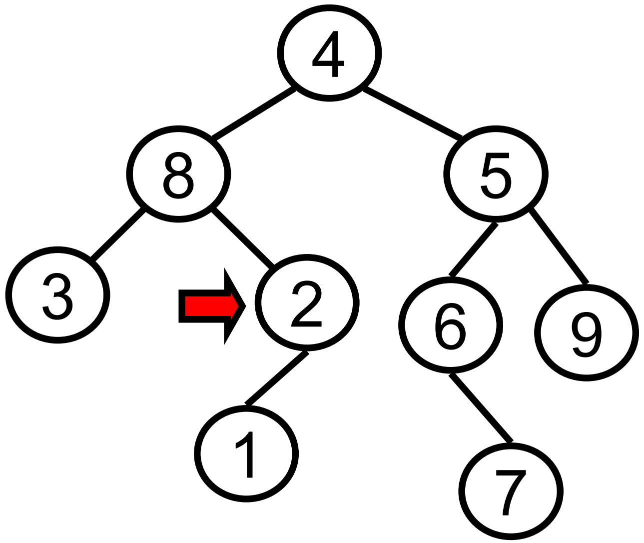
מעקב



L ➔ R ➔ Root : 3

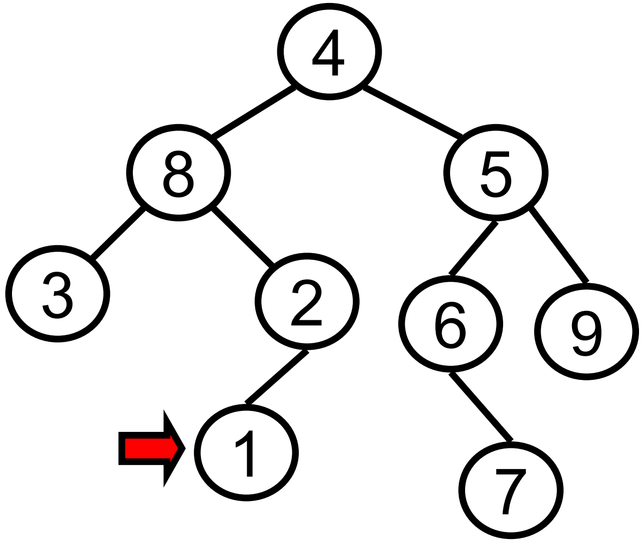


L ➔ R ➔ Root : 3

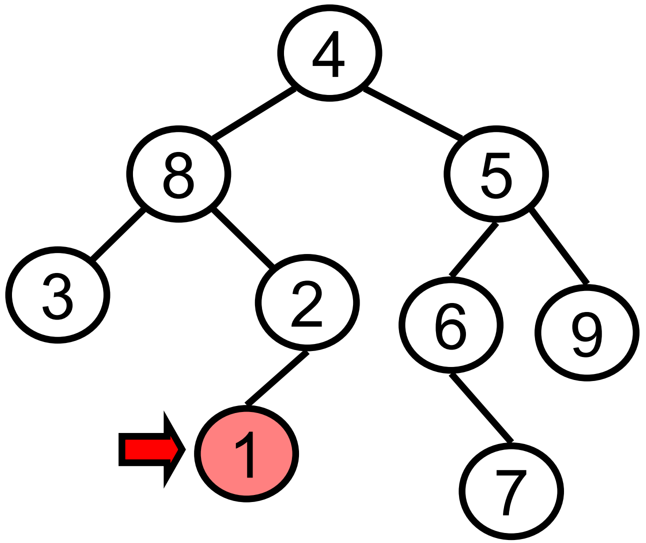


L ➔ R ➔ Root : 3

מעקב

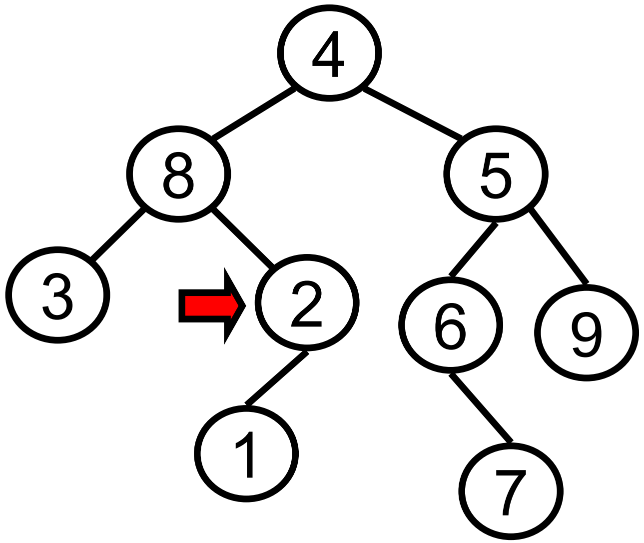


L ➔ R ➔ Root : 3

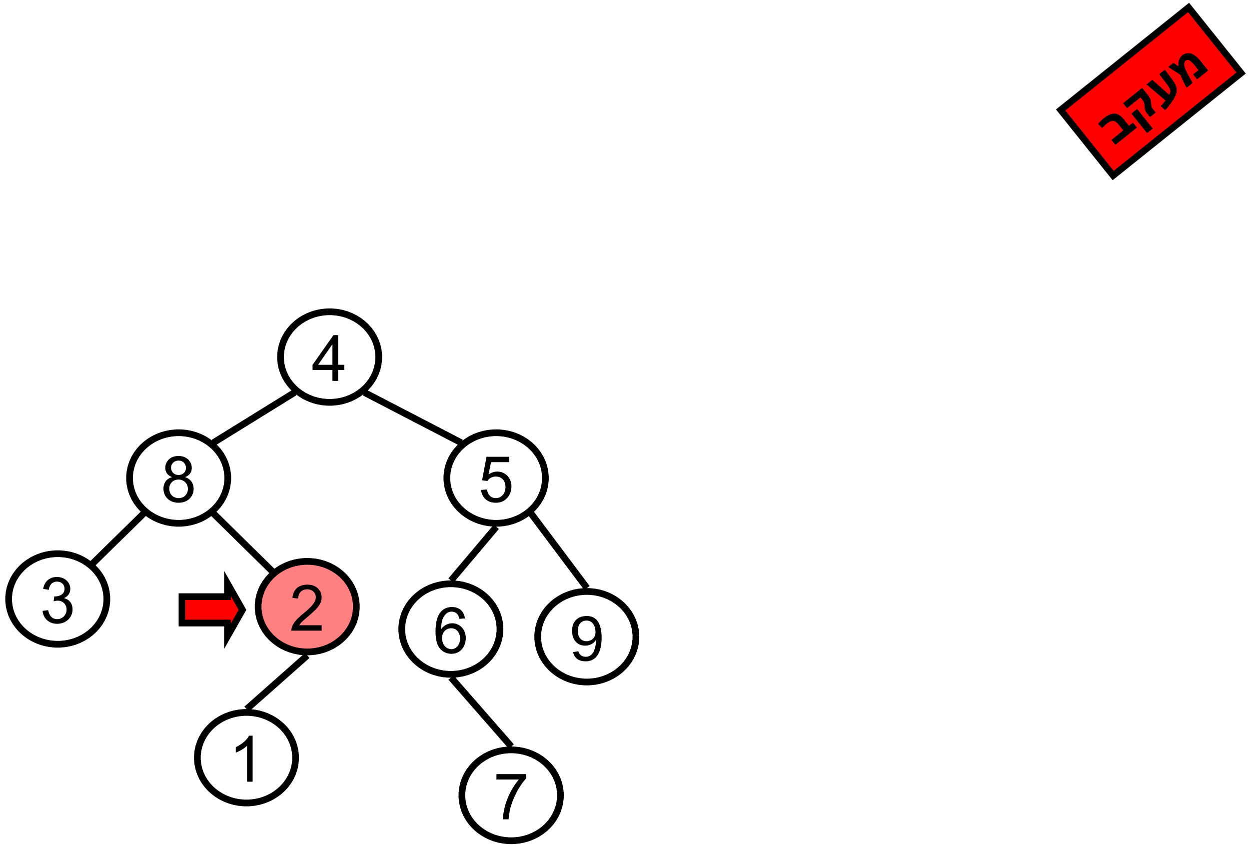


L ➔ R ➔ Root : 3 1

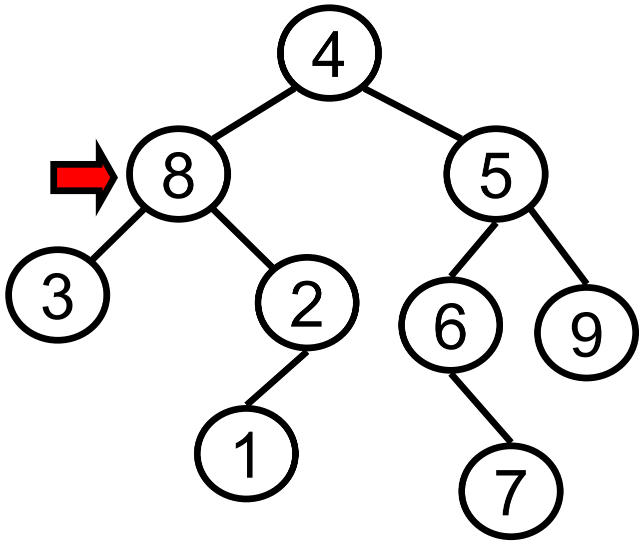
מעקב



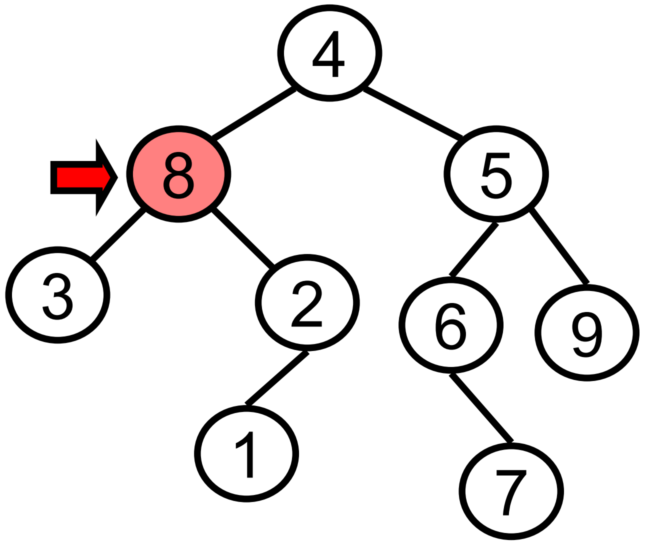
L ➔ R ➔ Root : 3 1



L ➔ R ➔ Root : 3 1 2

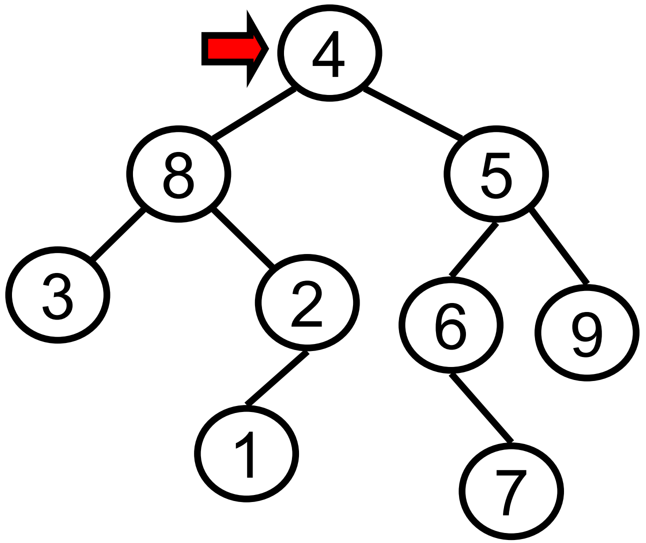


L ➔ R ➔ Root : 3 1 2

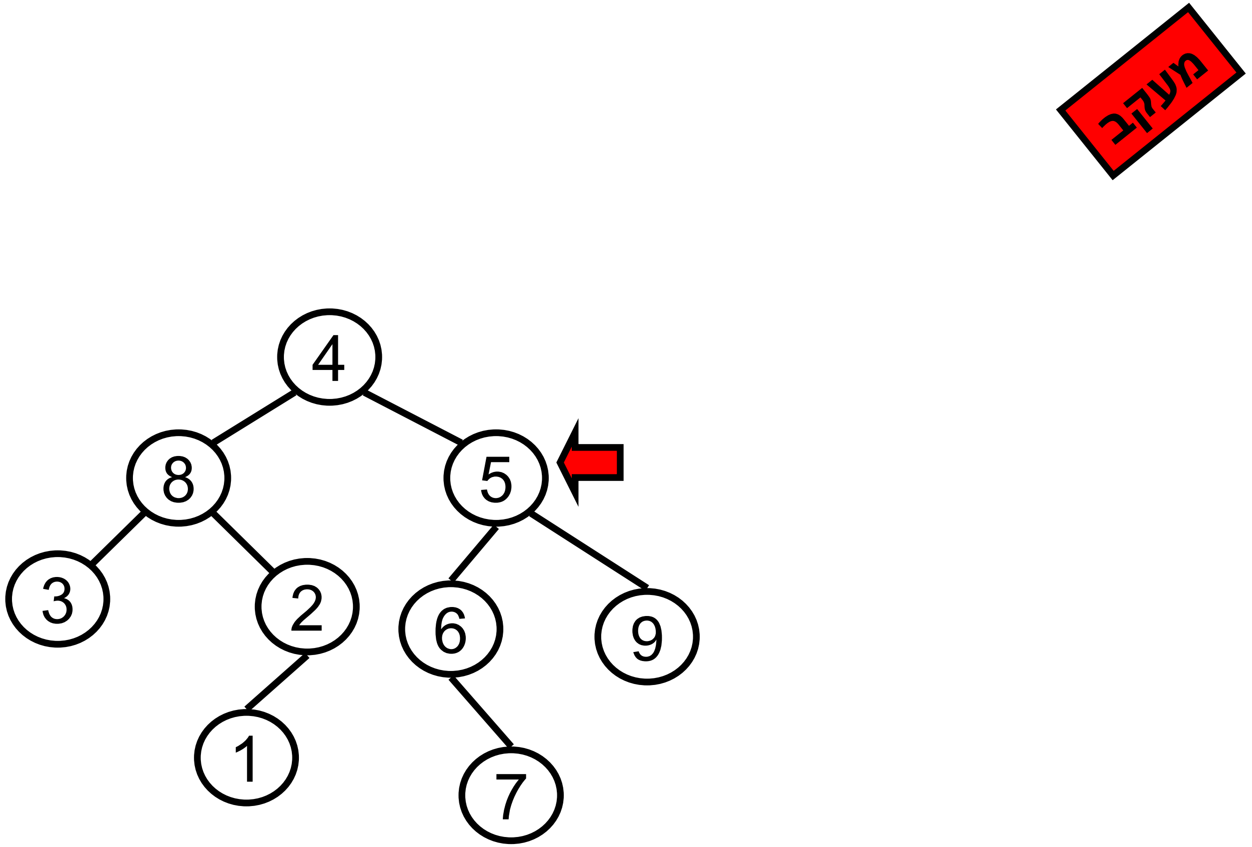


L ➔ R ➔ Root : 3 1 2 8

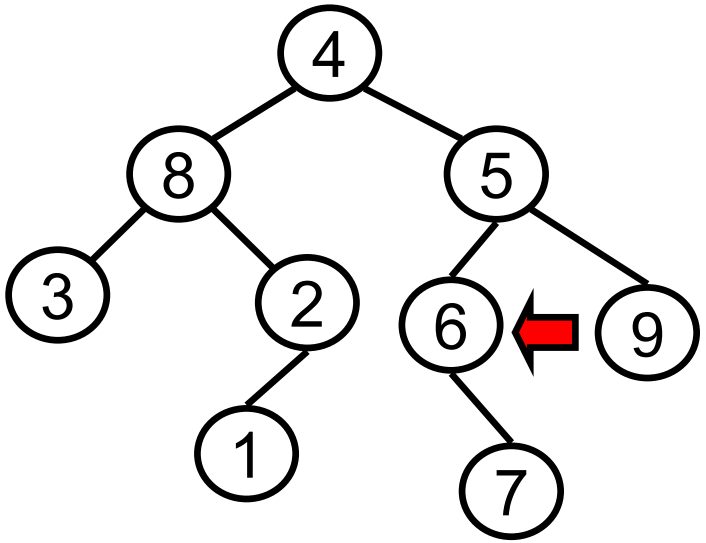
מעקב



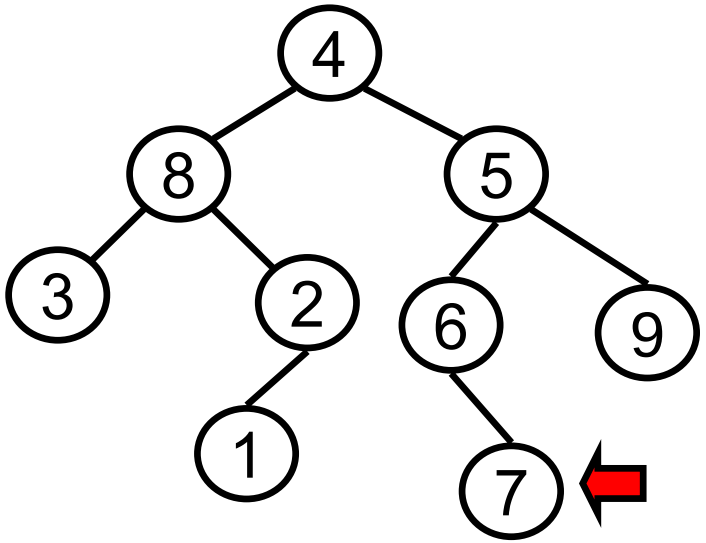
L ➔ R ➔ Root : 3 1 2 8



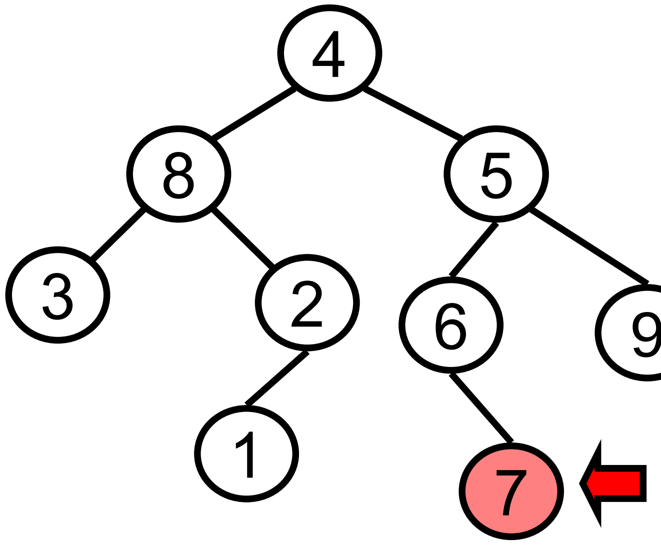
L ➔ R ➔ Root : 3 1 2 8



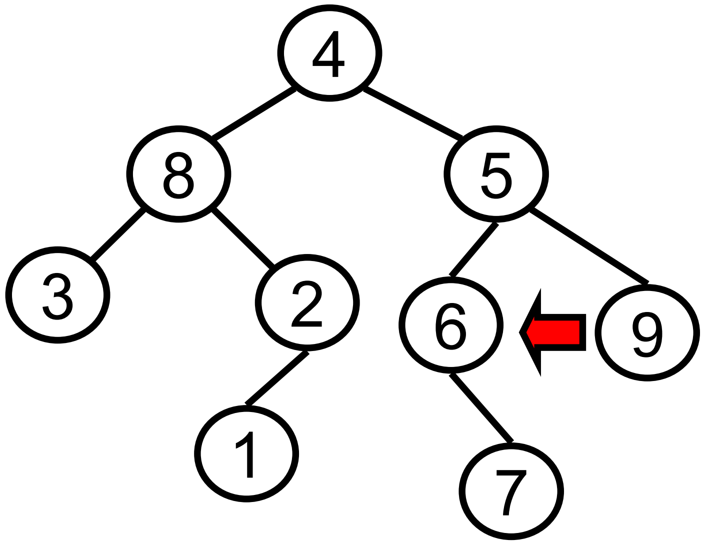
L ➔ R ➔ Root : 3 1 2 8



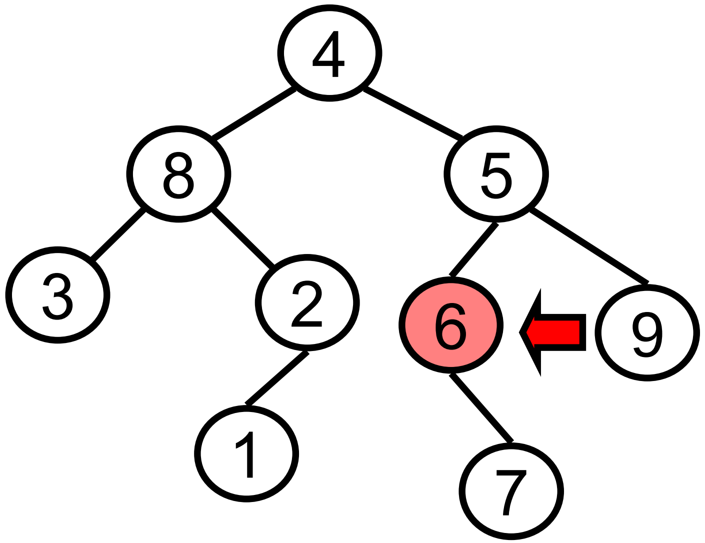
L ➔ R ➔ Root : 3 1 2 8



L ➔ R ➔ Root : 3 1 2 8 7

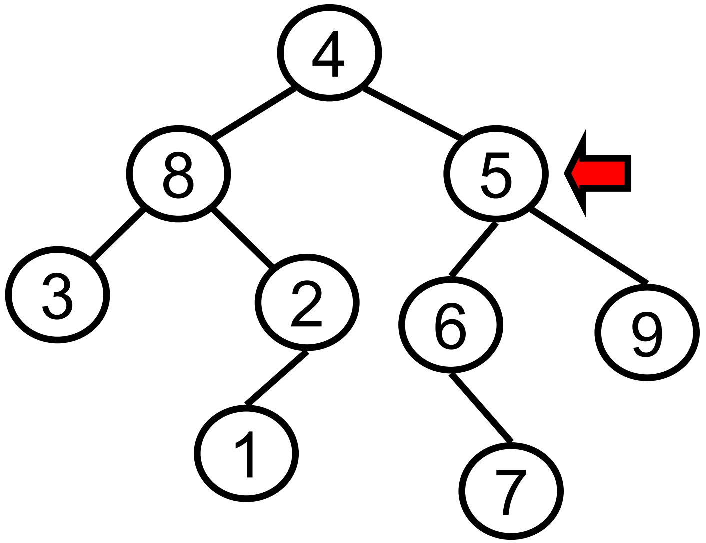


L ➔ R ➔ Root : 3 1 2 8 7

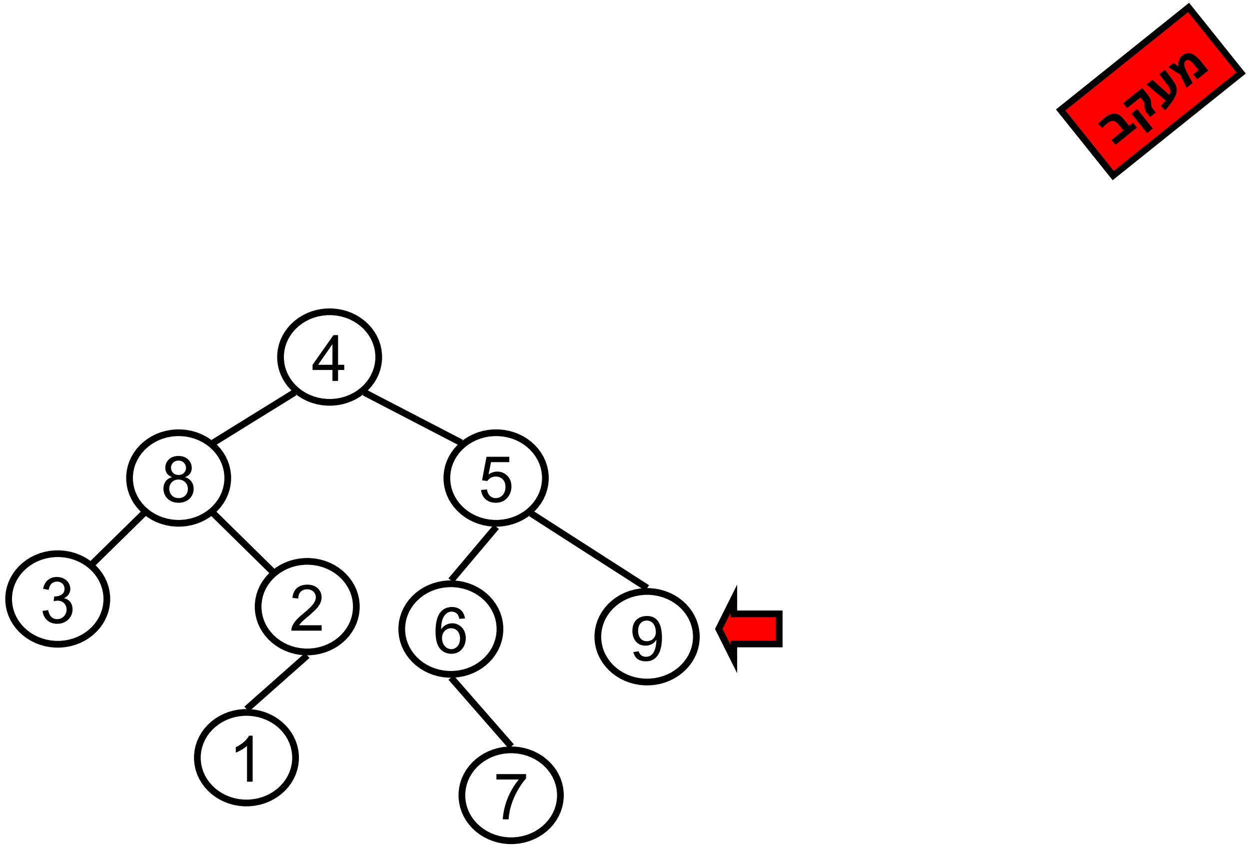


L ➔ R ➔ Root : 3 1 2 8 7 6

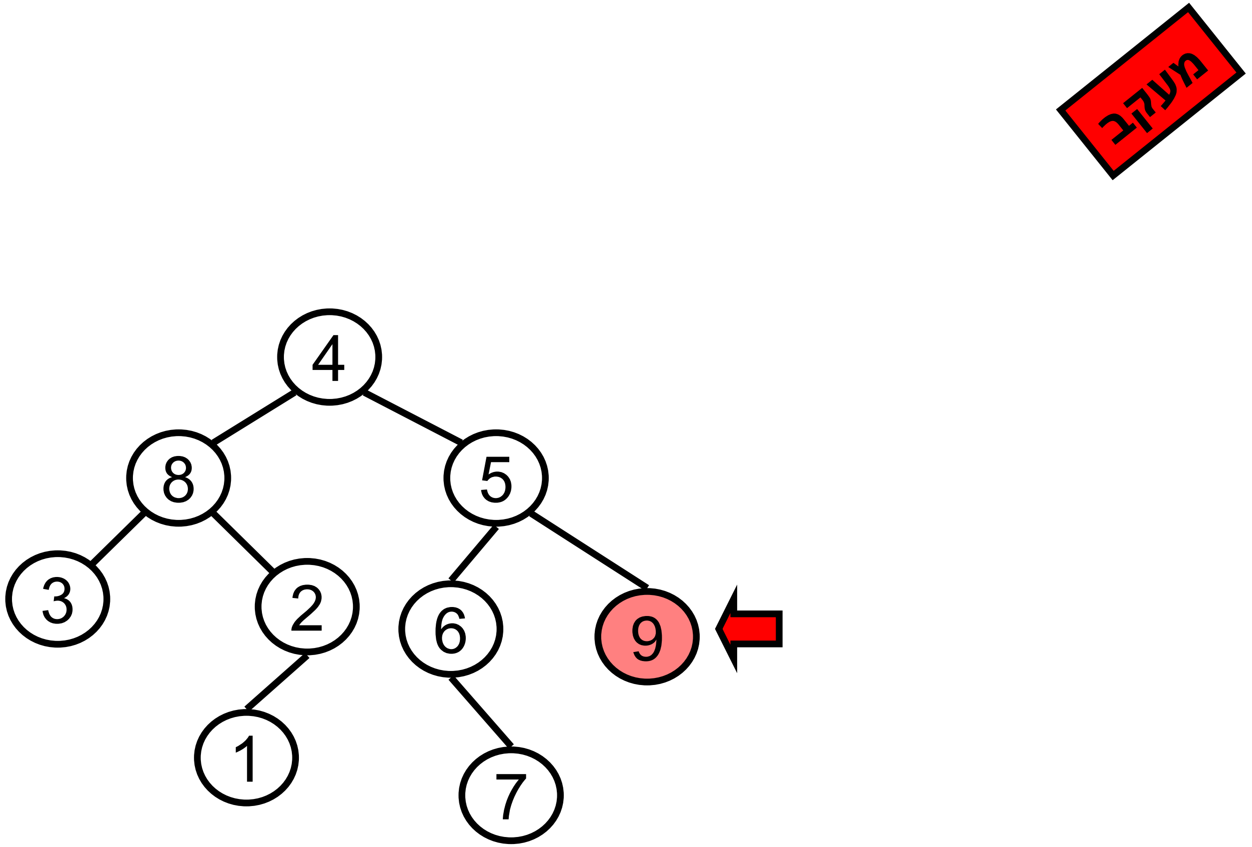
מעקב



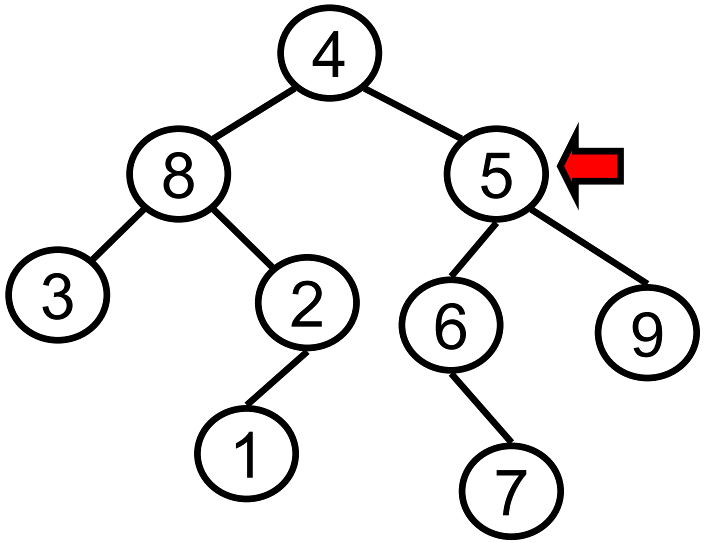
L ➔ R ➔ Root : 3 1 2 8 7 6



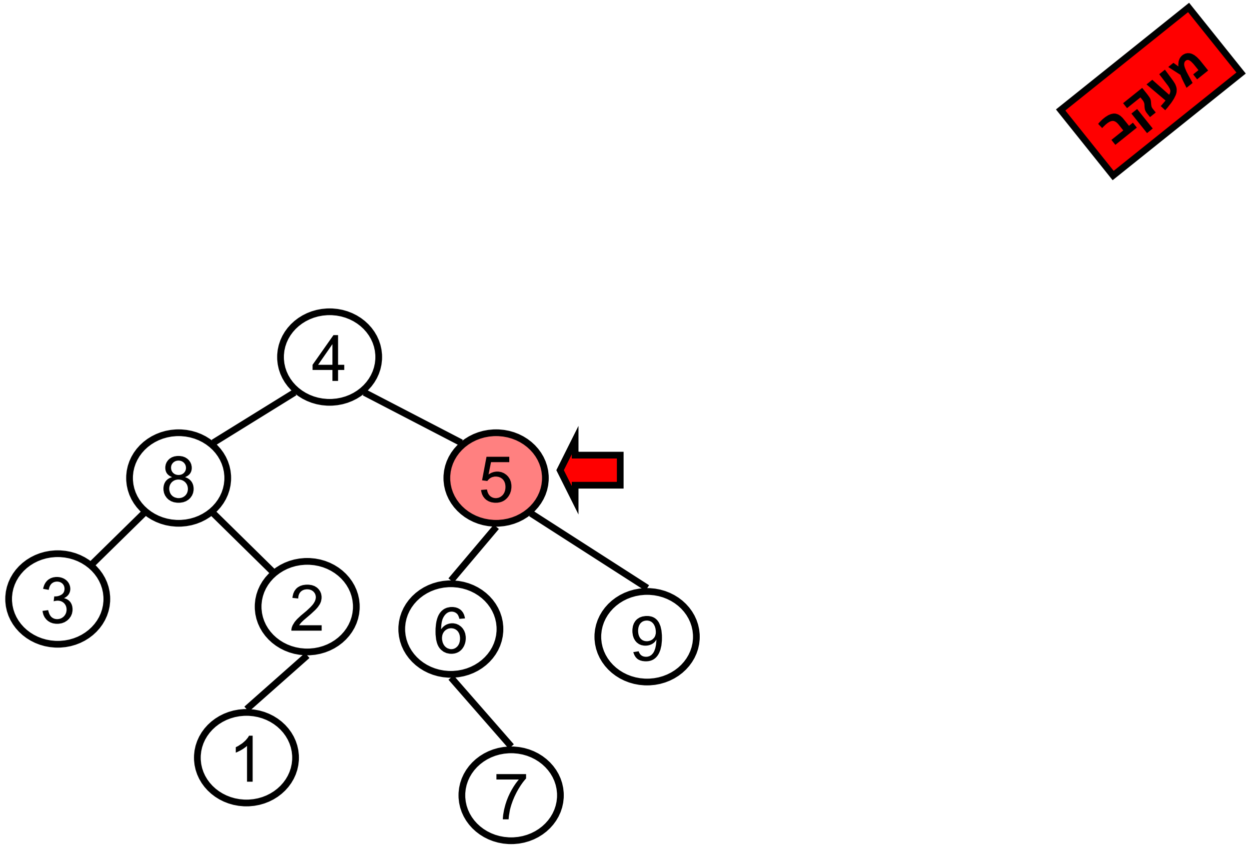
L ➔ R ➔ Root : 3 1 2 8 7 6



L ➔ R ➔ Root : 3 1 2 8 7 6 9

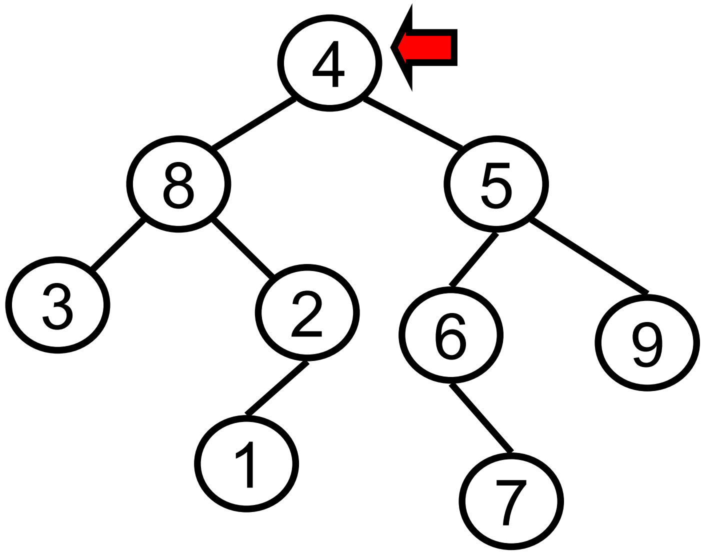


L ➔ R ➔ Root : 3 1 2 8 7 6 9



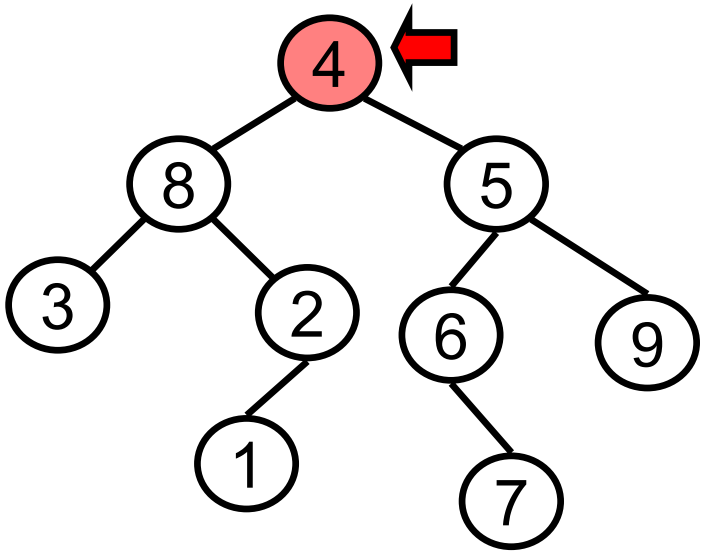
L ➔ R ➔ Root : 3 1 2 8 7 6 9 5

מעקב



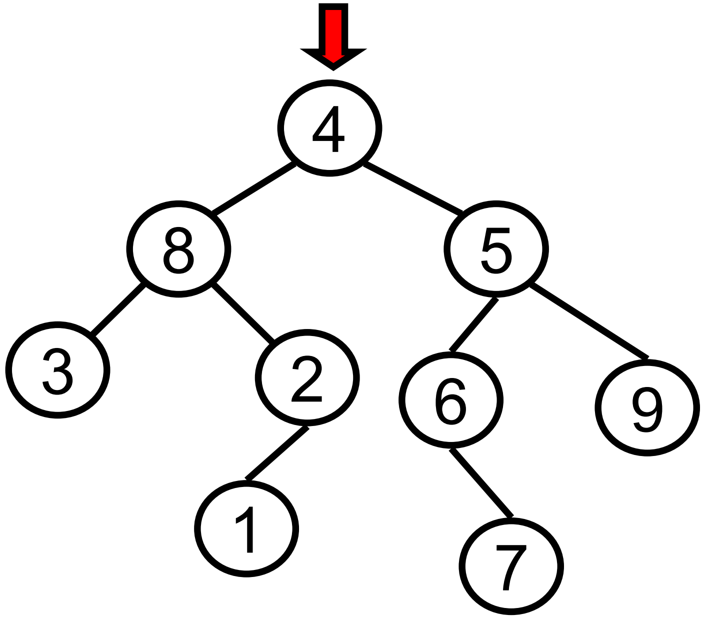
L ➔ R ➔ Root : 3 1 2 8 7 6 9 5

מעקב



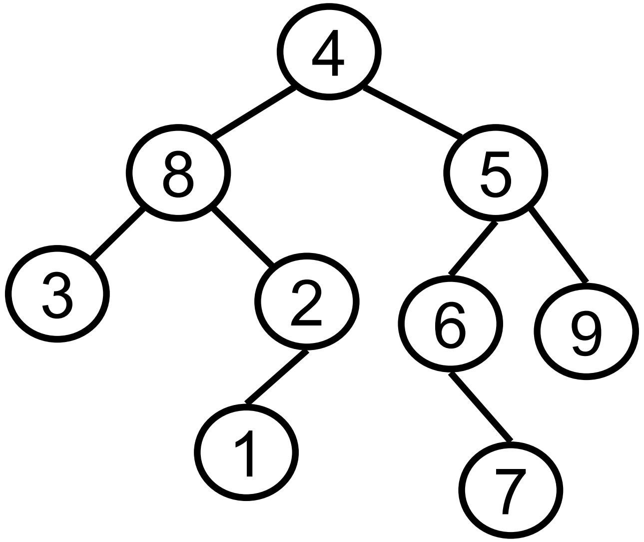
L ➔ R ➔ Root : 3 1 2 8 7 6 9 5 4

מעקב

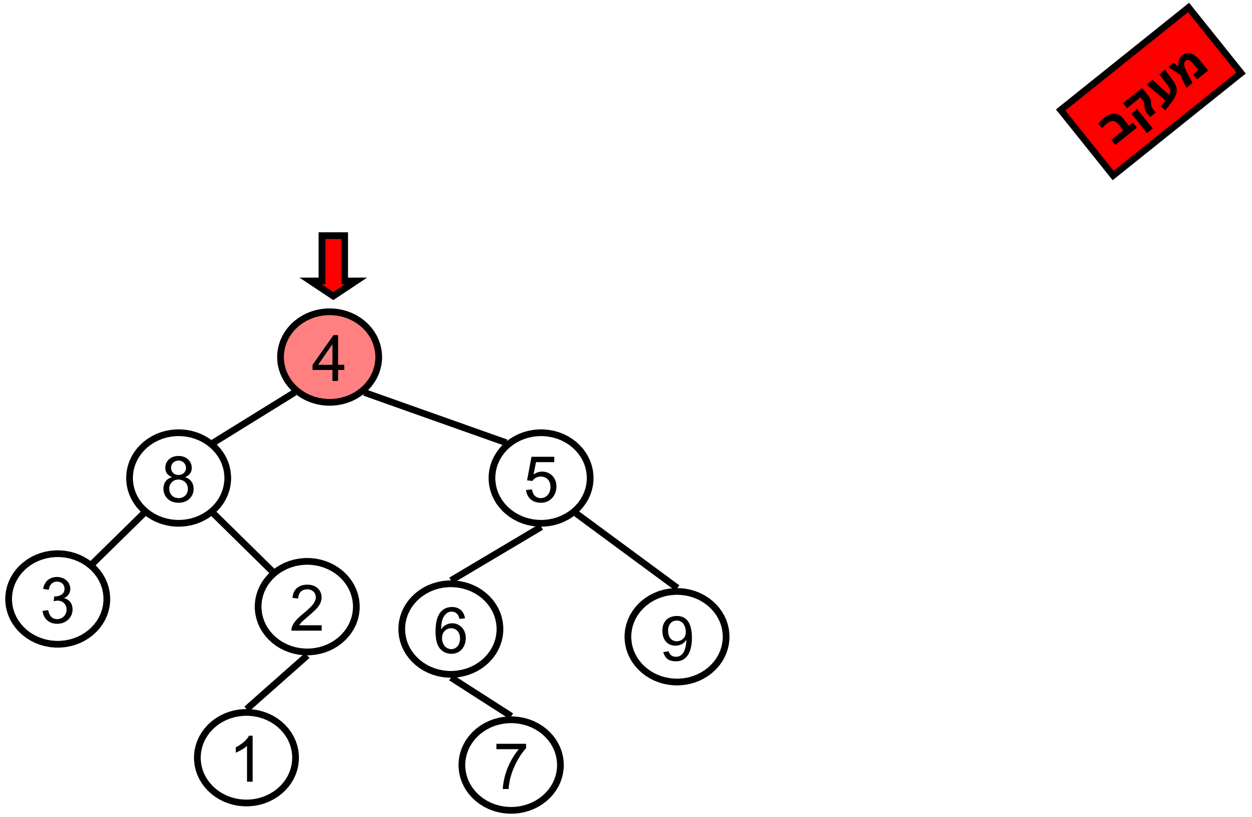


L ➔ R ➔ Root : 3 1 2 8 7 6 9 5 4

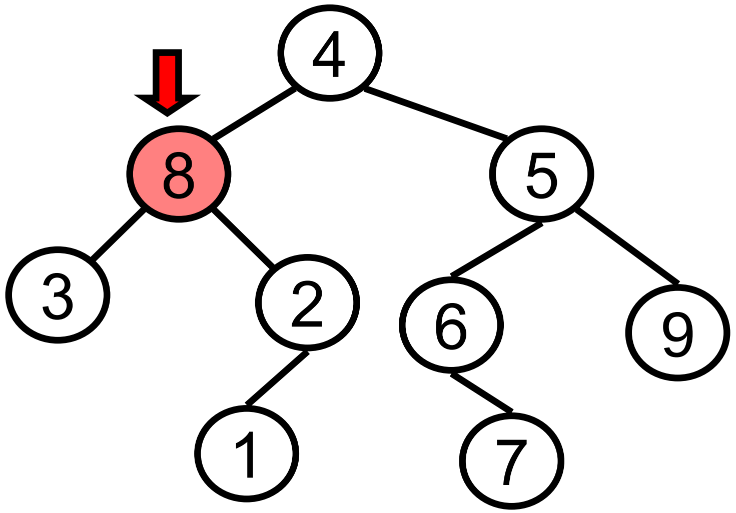
Breadth First Traversal
סריקה לרוחב



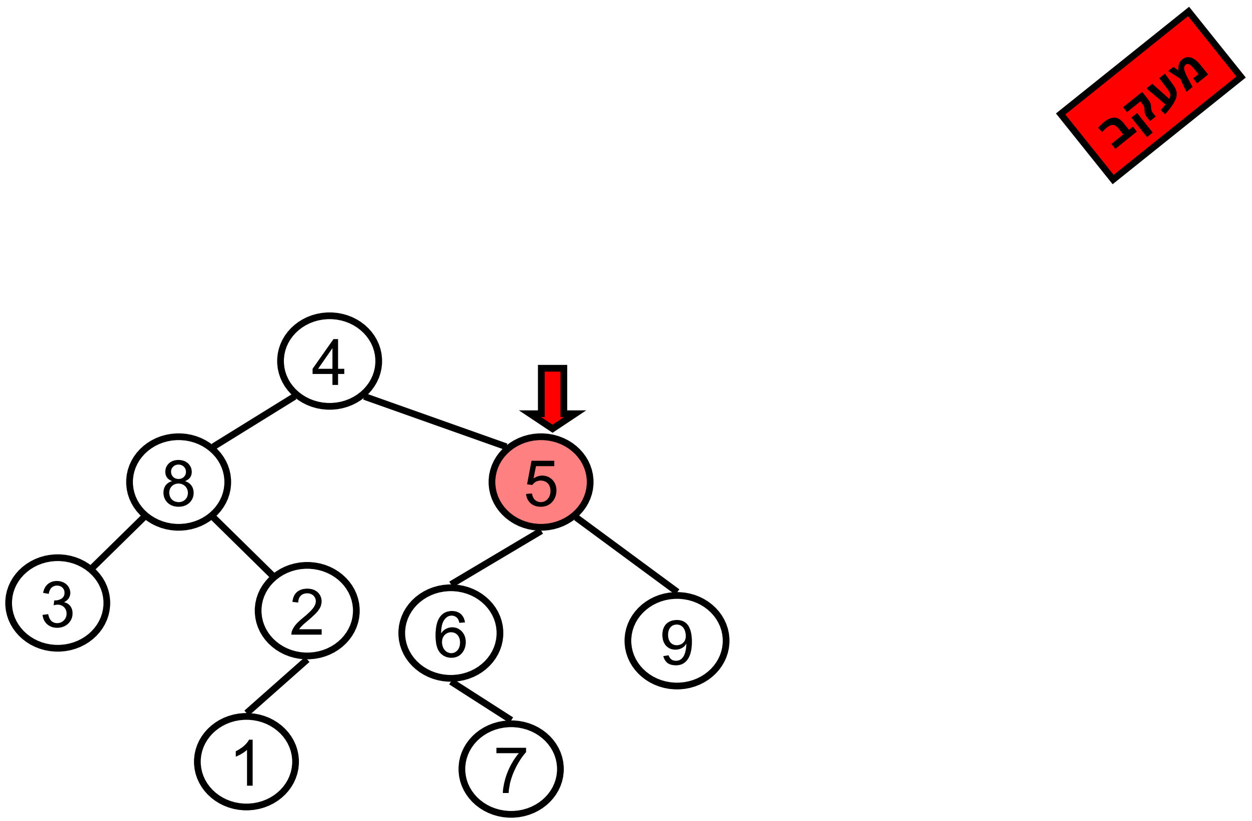
LevelOrder ➔ 4 8 5 3 2 6 9 1 7



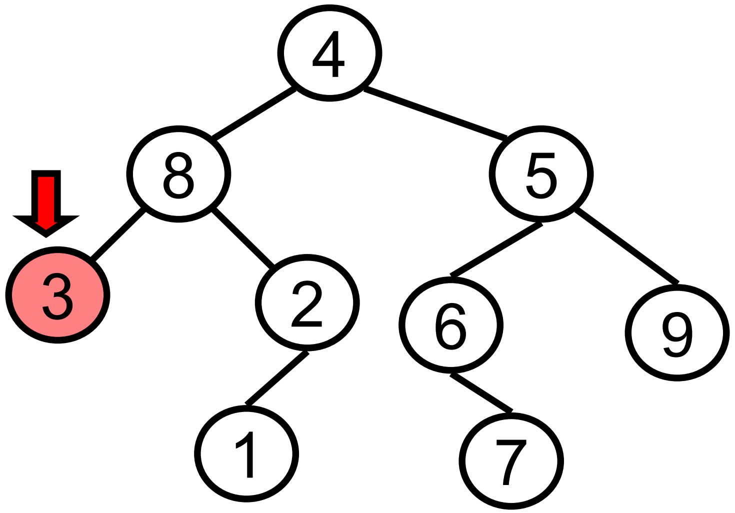
LevelOrder ➔ 4



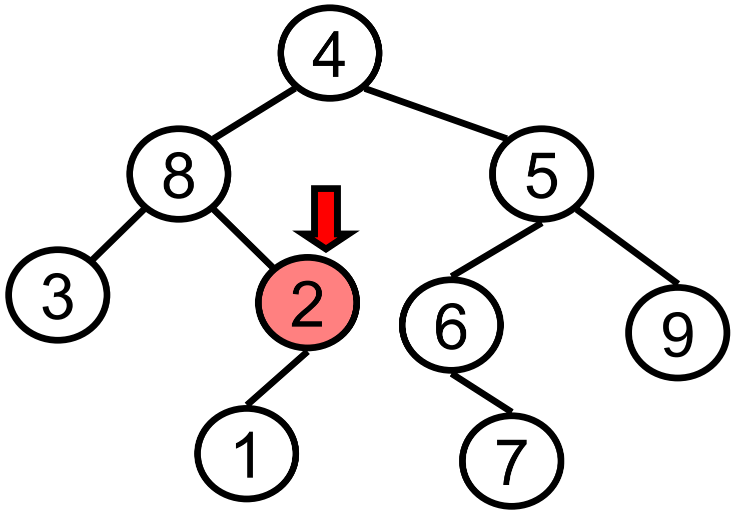
LevelOrder ➔ 4 8



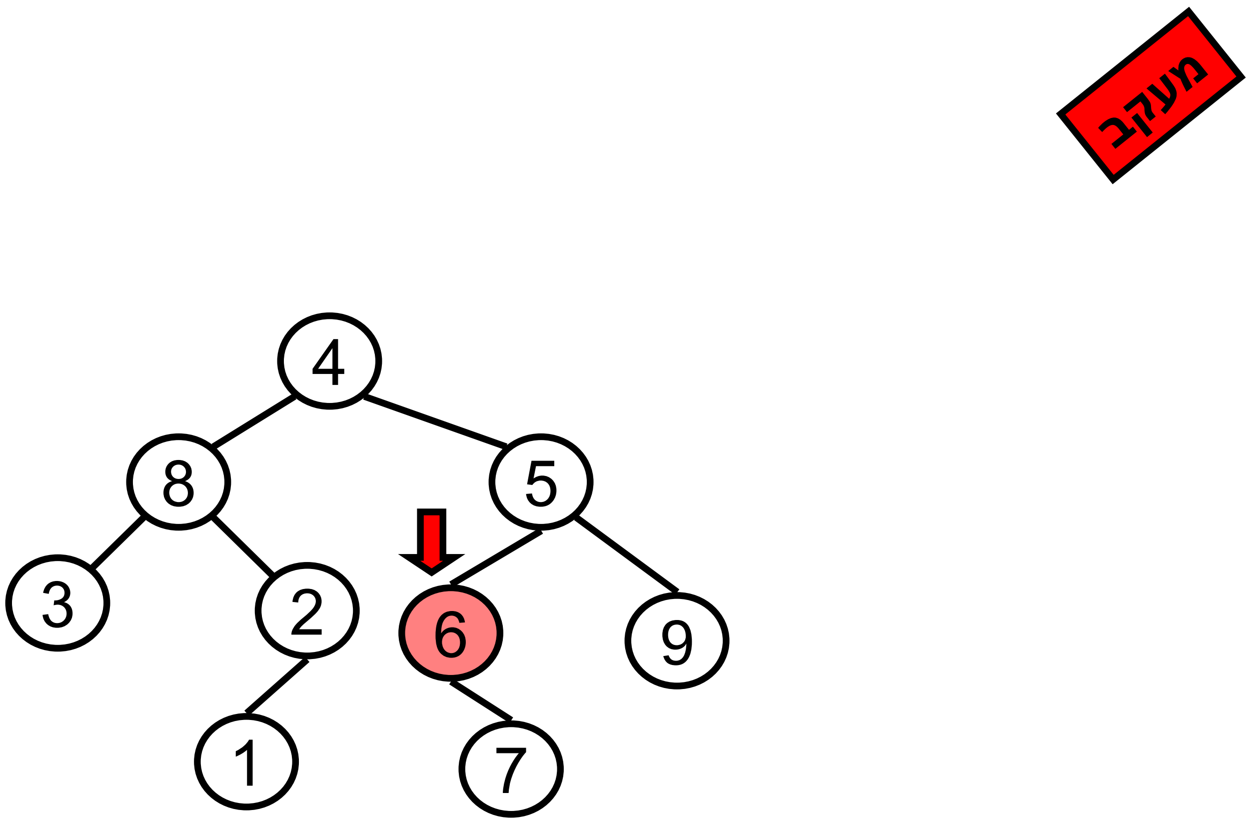
LevelOrder ➔ 4 8 5



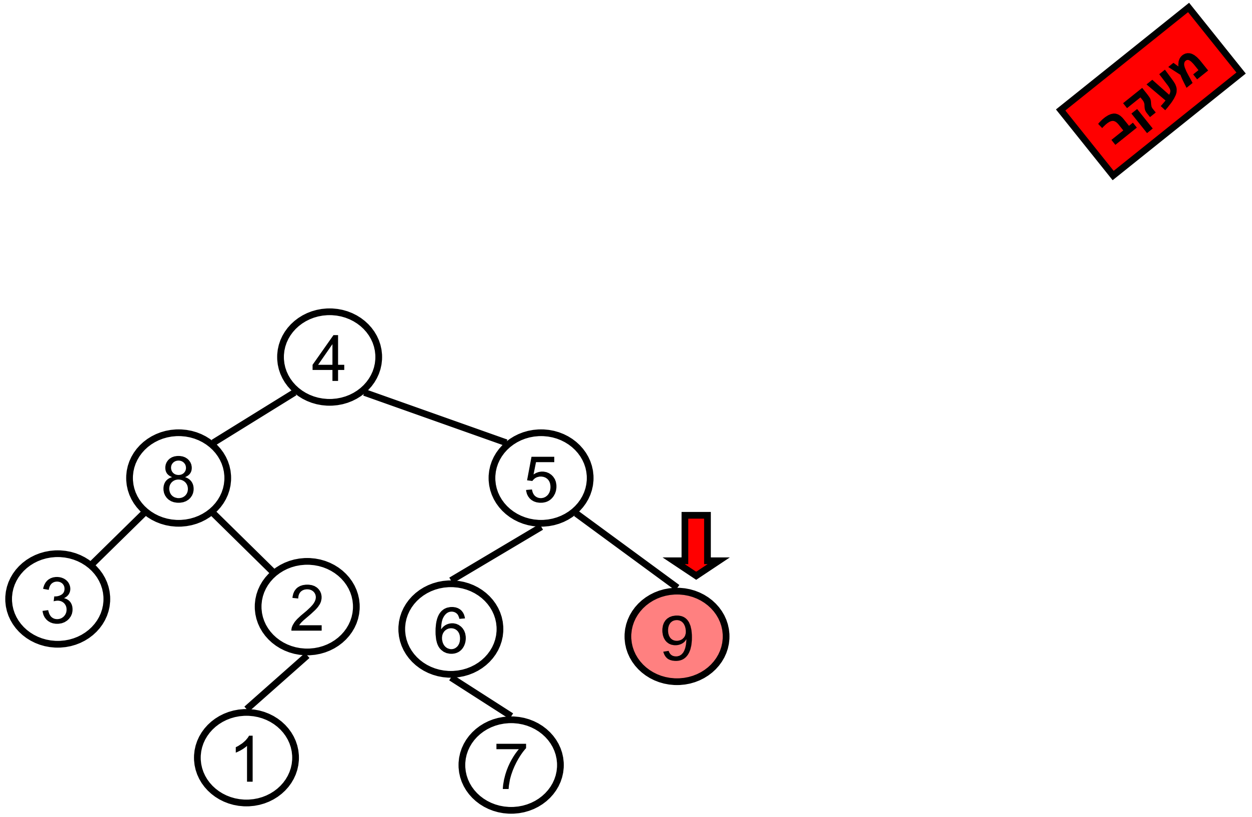
LevelOrder ➔ 4 8 5 3



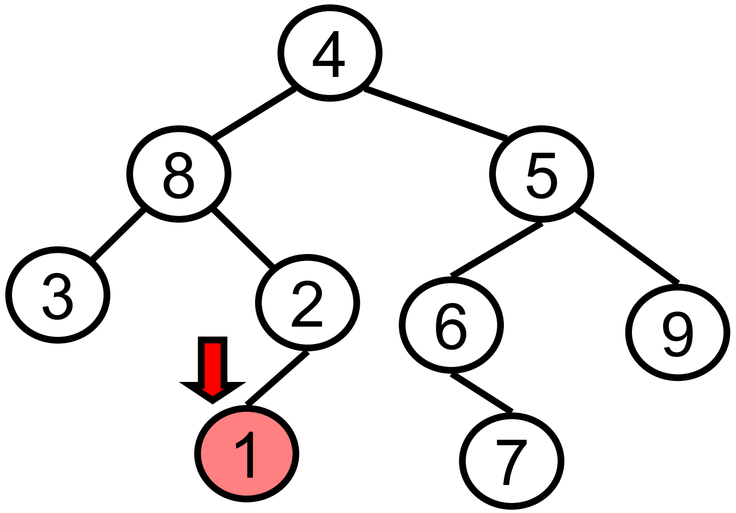
LevelOrder ➔ 4 8 5 3 2



LevelOrder ➔ 4 8 5 3 2 6

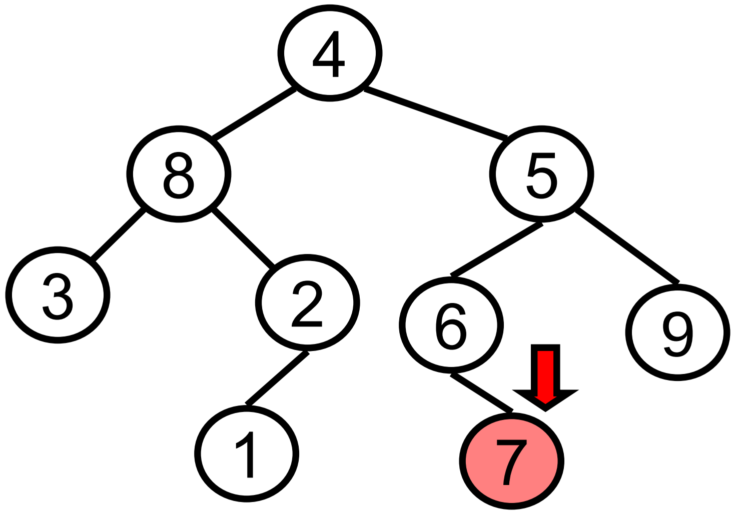


LevelOrder ➔ 4 8 5 3 2 6 9

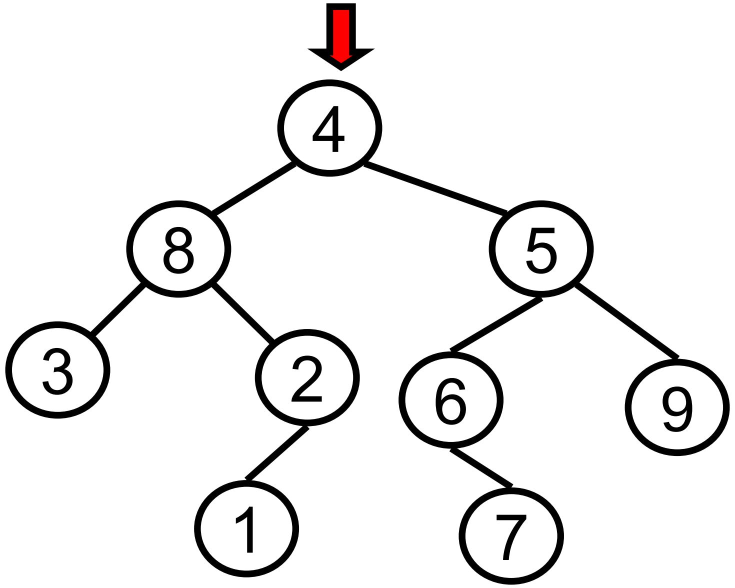


LevelOrder ➔ 4 8 5 3 2 6 9 1

מעקב

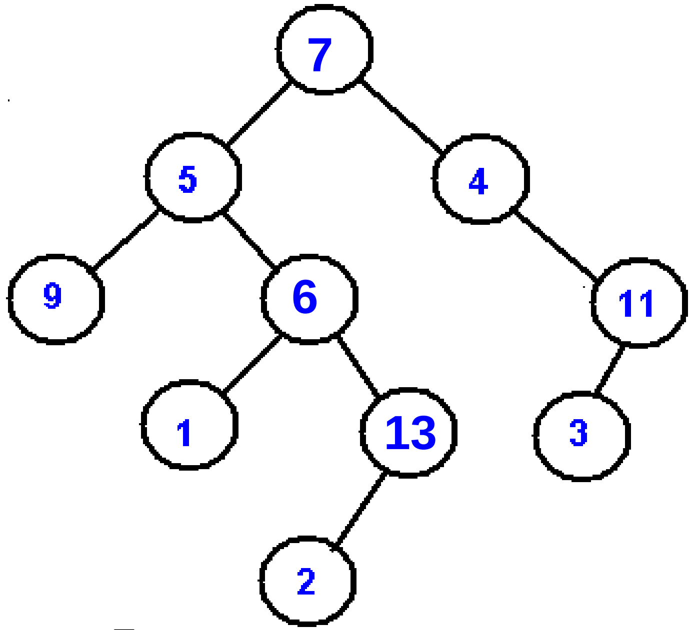


LevelOrder → 4 8 5 3 2 6 9 1 7



LevelOrder ➔ 4 8 5 3 2 6 9 1 7

עבור העץ הבינארי הבא מה יודפס אם מתבצעת סריקה
1. תחילית 2. סופית 3. תוכית 4. רוחבית

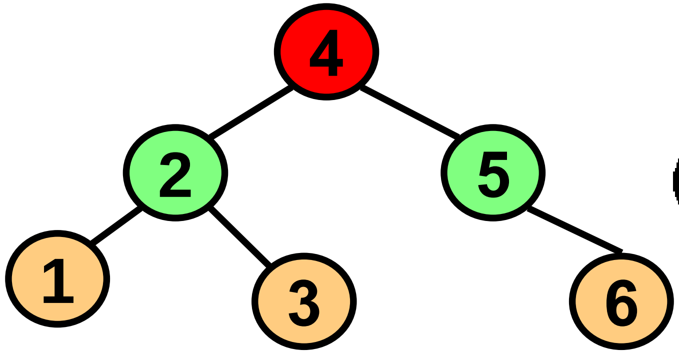


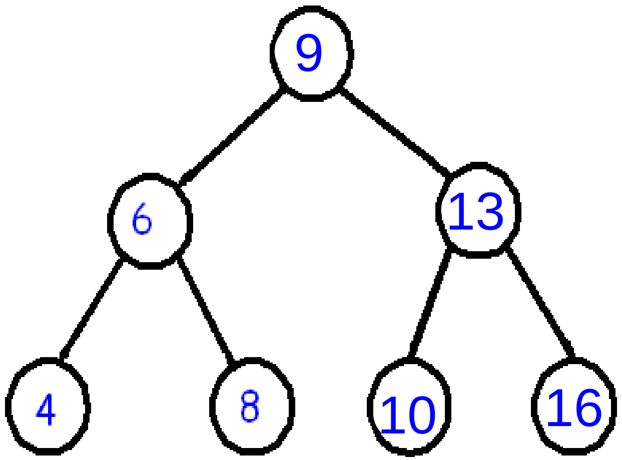
PreOrder ➔ 7, 5, 9, 6, 1, 13, 2, 4, 11, 3
PostOrder ➔ 9, 1, 2, 13, 6, 5, 3, 11, 4, 7
InOrder ➔ 9, 5, 1, 6, 2, 13, 7, 4, 3, 11
LevelOrder ➔ 7, 5, 4, 9, 6, 11, 1, 13, 3, 2

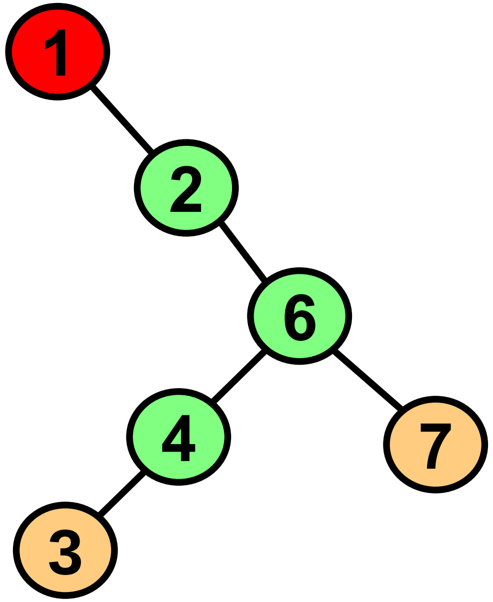
Binary Search Tree (BST)
עץ חיפוש בינארי

יהי x צומת כלשהו בעץ חיפוש בינארי אזי מתקיים :

1. לכל צומת עבתת העץ השמאלי של x מתקיים ש $key[y] < key[x]$
2. לכל צומת עבתת העץ הימני של x מתקיים ש $key[x] \leq key[y]$

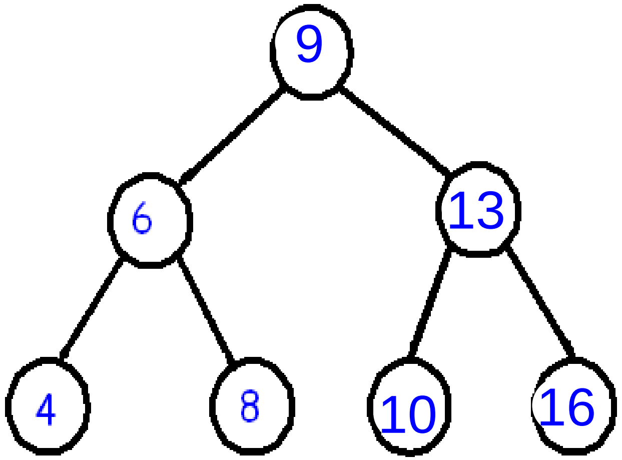






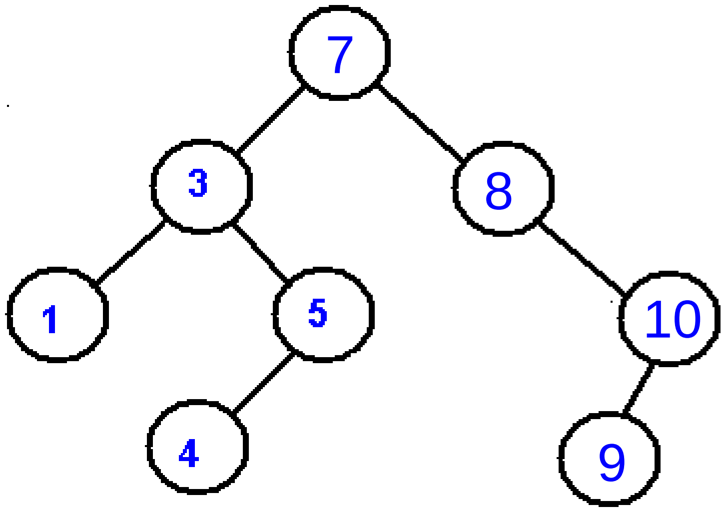
חיפוש איבר
בעץ חיפוש בינארי

O(h)
best case O(log(n))
worst case O(n)

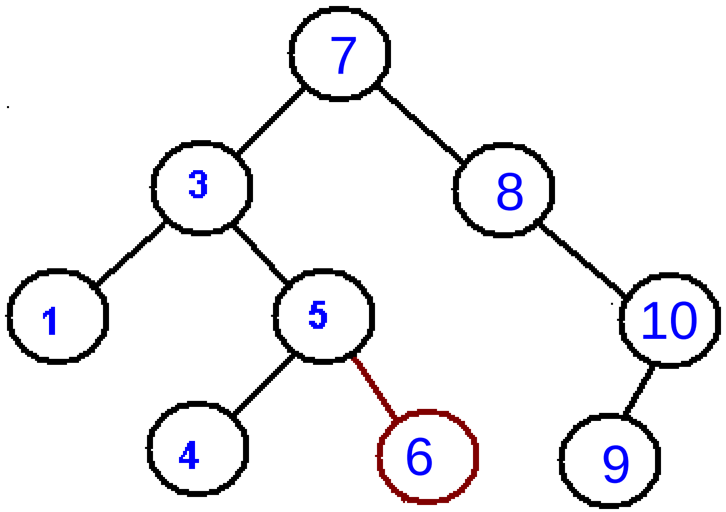


יהי x האיבר אותו מחפשים
מתחילים בשורש
אם x שווה לאיבר בשורש עוצרים
אם קטן עוברים לתת העץ השמאלי ומשם מבצעים סריקה מחדש
אם גדול עוברים לתת העץ הימני ומשם מבצעים סריקה מחדש

הוספת איבר
בעץ חיפוש בינארי



לפני הוספת 6



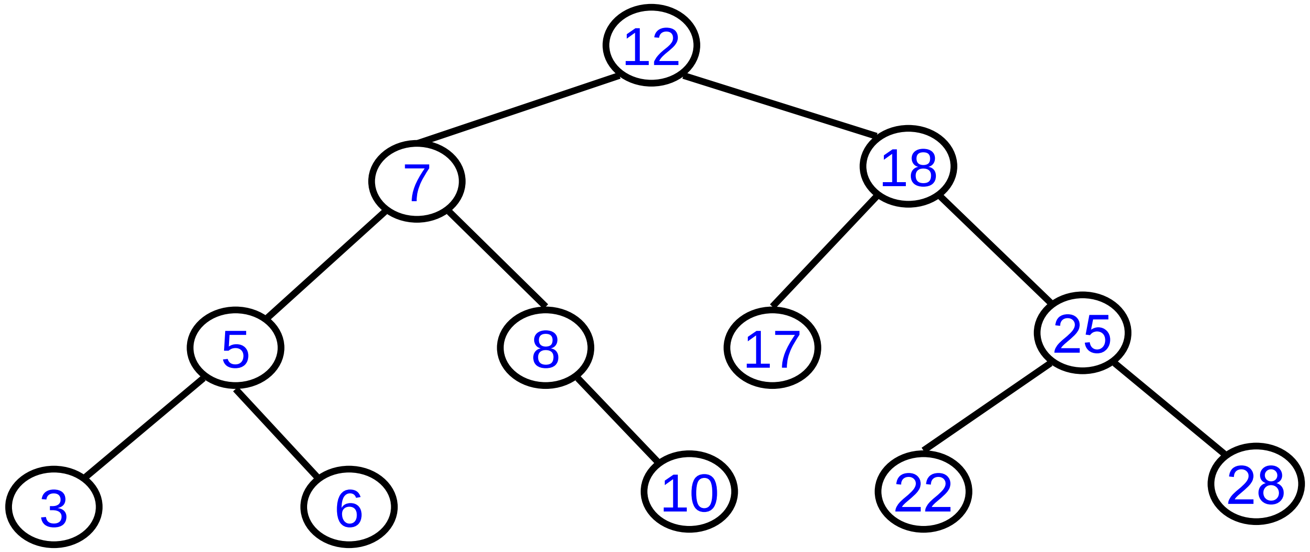
אחרי הוספת 6

חסרון של שיטה זו הוא
שהעץ המתקבל לא מאוזן

תהליך ההוספה דומה לתהליך החיפוש. מתחילים את החיפוש בשורש עד שמוצאים את המקום. אם האיבר קיים לא מוספים. אחרת מוסיפים את האיבר.

בנה עץ חיפוש בינארי
ע"י הכנסת האיברים שלמטה משמאל לימין

→ **12, 7, 8, 18, 5, 10, 6, 17, 25, 28, 22,3**



הסרת איבר
מעץ חיפוש בינארי

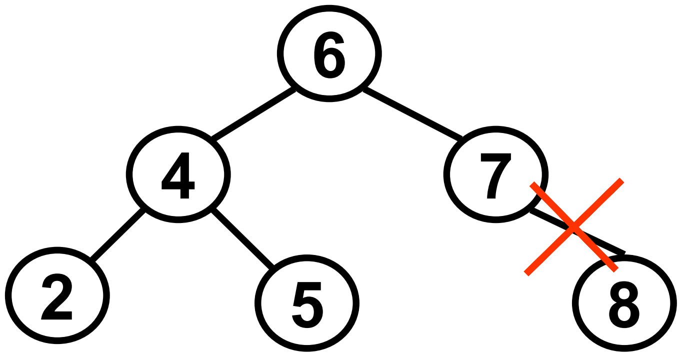
ארבע מקרים שונים

- האיבר לא קיים
- האיבר עלה
- לאיבר יש בן יחיד
- לאיבר יש שני בנים

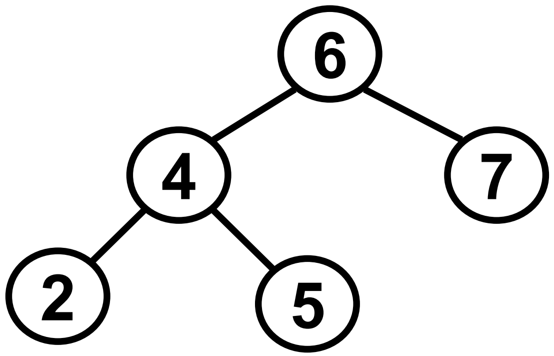
ההסרה דומה להסרה
מרשימה מקושרת רגילה

הסרת עלה
מעץ חיפוש בינארי

רוצים להסיר 8 מהעץ



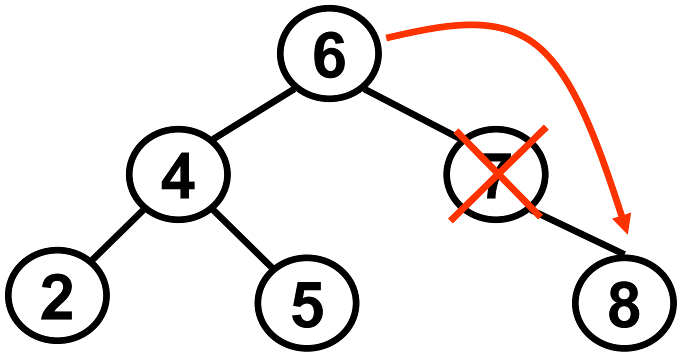
לפני ההסרה



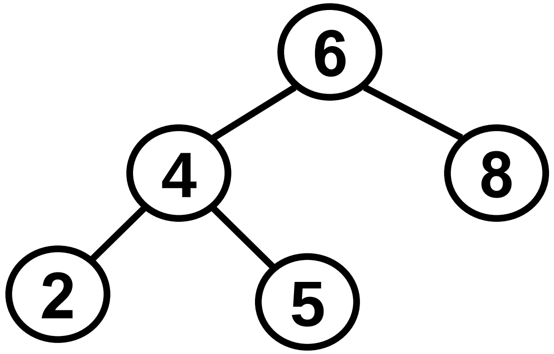
אחרי ההסרה

הסרת איבר בעל בן יחיד
מעץ חיפוש בינארי

רוצים להסיר 7 מהעץ
מעבירים את המצביע לבן



לפני ההסרה

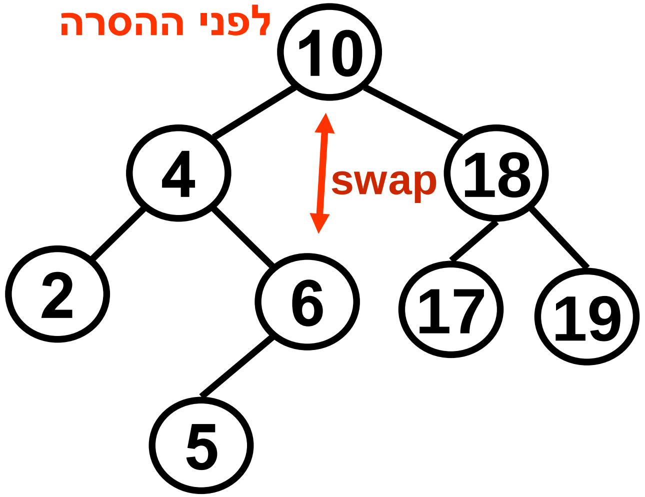


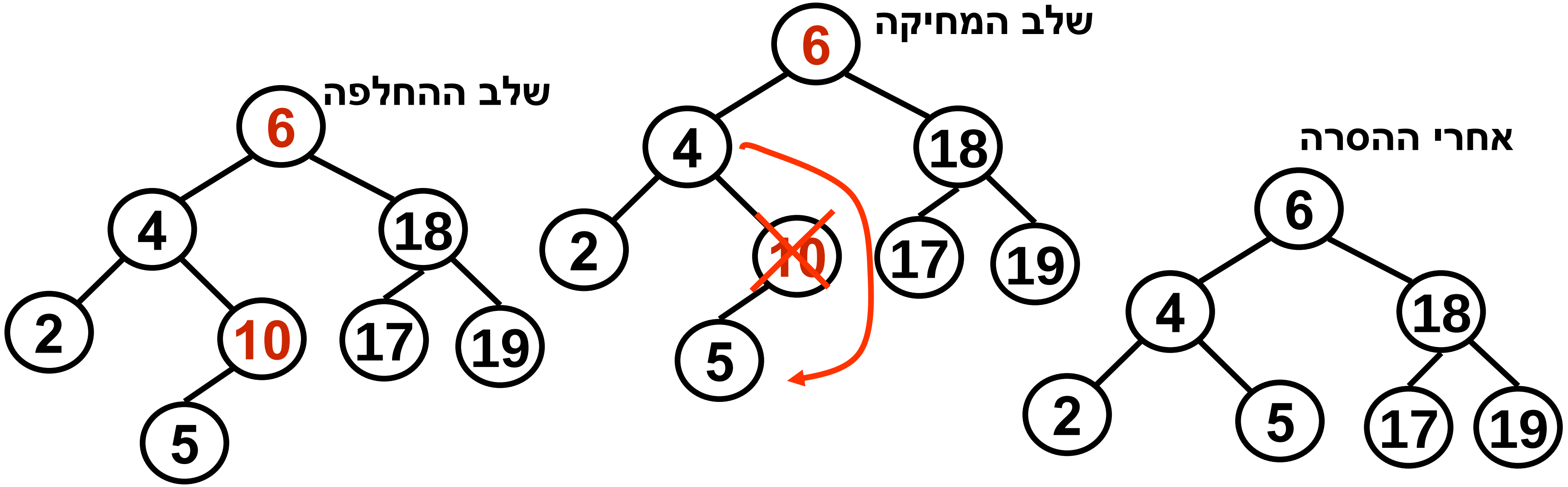
אחרי ההסרה

הסרת איבר בעל שני בנים
מעץ חיפוש בינארי

רוצים להסיר 10 מהעץ
מחליפים עם האיבר הגדול ביותר בתת העץ השמאלי
(או עם הקטן ביותר בתת העץ הימני) (6 או 17)
התוצאה: או עלה להסרה או אבא לבן יחיד

לפני ההסרה





נתון המבנה Node אשר
מתאר קודקוד בעץ בינארי

```
typedef struct Node
{
    int num;
    struct Node* left;
    struct Node* right;
} Node;
```

כתוב את התוכנית BST.c אשר מתארת עץ חיפוש בינארי
ומיישמת את הפעולות הבאות :

BST.h

```
typedef struct Node {
    int num;
    struct Node* left;
    struct Node* right;
} Node;
Node* newNode(int x);
void insert(Node** root, int x);
void destroyTree(Node *root);
Node* search(Node *root, int x);
Node* maxNode(Node* tmp);
int degree (Node* temp);
int height (Node* root);
void preorder(Node *root);
void inorder(Node *root);
void postorder(Node *root);
void removeNode(Node** root, int num);
void removeLeaf (Node** root, Node* leaf);
void removeParentOfOneChild (Node** root, Node* node);
void removeParentOfTwoChildren(Node** root, Node* node);
Node* parent(Node* root, Node* node);
void replace(Node* tmp1,Node* tmp2 );s
```

main.c

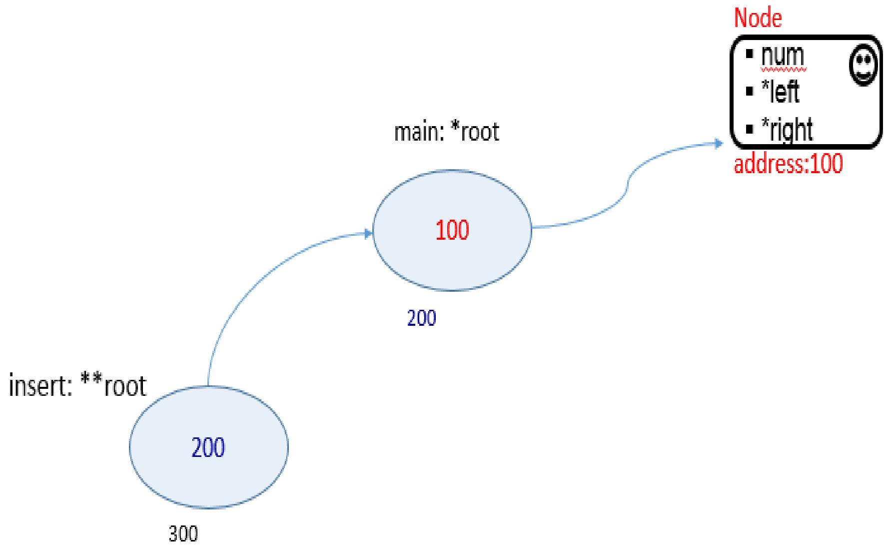
```c
#include <stdio.h>
#include "BST.h"
int main() {
    int arr[ ] = {12, 7, 8, 18, 5, 10, 6, 17, 25, 28, 22,3},i;
    Node* root=NULL;

    for(i=0; i<12; i++)
        insert(&root,arr[i]);

    preorder(root);
    printf("\n%p\n",(void*)search(root,8));
    printf("%d\n",maxNode(root)->num);
    printf("%d\n",degree(root));
    printf("%d\n",height(root));
    removeNode(&root,17);
    removeNode(&root,18);
    removeNode(&root,5);
    removeNode(&root,12);
    preorder(root);
    putchar('\n');
    destroyTree(root);
    root=NULL;
    return 0;
}
```

Output:

```
12 7 5 3 6 8 10 18 17 25 22 28
0x8209028
28
2
3
10 7 3 6 8 25 22 28
```




```c
Node* newNode(int x)
{
    Node *tmp;
    tmp = (Node *) malloc(sizeof(Node));
    if(tmp != NULL)
    {
        tmp->num=x;
        tmp->left = NULL;
        tmp->right = NULL;
    }
    return tmp;
}
```

BST.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "BST.h"
void insert(Node **root, int x)
{
    if( *root == NULL )
    {
        *root = newNode(x);
    }
    else if(x < (*root)->num)
    {
        insert(&(*root)->left, x);
    }
    else if(x > (*root)->num)
    {
        insert(&(*root)->right, x);
    }
}
:
```

```c
void destroyTree(Node *root)
{
    if( root == NULL )
        return;
    destroyTree(root->left);
    destroyTree(root->right);
    free(root);
}
```

BST.c cont.

```
void preorder(Node *root)
{
    if(root == NULL)
        return;
    printf("%d ", root->num);
    preorder(root->left);
    preorder(root->right);
}
```

```
void inorder(Node *root)
{
    if (root == NULL)
        return;
    inorder(root->left);
    printf("%d ", root->num);
    inorder(root->right);
}
```

```
void postorder(Node *root)
{
    if (root == NULL)
        return;
    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->num);
}
```

```
Node* maxNode(Node* tmp)
{
    while(tmp != NULL && tmp->right!=NULL)
    {
        tmp = tmp->right;
    }
    return tmp;
}
```

version1

```
Node* maxNode(Node* tmp)
{
    if(tmp == NULL || tmp->right==NULL)
        return tmp;

    return maxNode(tmp->right);
}
```

version2

```
int degree (Node* temp)
{
    if(temp->left !=NULL && temp->right!=NULL)
            return 2;
    else if(temp->left!=NULL || temp->right!=NULL)
            return 1;
        return 0;
}
```

```
Node* search(Node *root, int x)
{
    while (root != NULL)
    {
        if (root->num == x)
            return root;
        else if (x < root->num)
            root = root->left;
        else
            root = root->right;
    }
    return NULL;
}
```

```
int height (Node* root)
{
    int a,b;
    if (root==NULL)
        return -1;
    a = height(root->left)+1;
    b = height(root->right)+1;
    return (a>b?a:b);
}
```

```
void swap(Node* tmp1,Node* tmp2 )
{
    if( tmp1==NULL || tmp2==NULL)
        return;
    int x = tmp2->num;
    tmp2->num = tmp1->num ;
    tmp1->num = x;
}
```

BST.c cont.

```
void removeNode (Node** root, int num)
{
    Node* tmp1 = search (*root, num);

    if(tmp1==NULL)
        return;

    if(degree(tmp1)==0)
        removeLeaf(root,tmp1);
    else if(degree(tmp1)==1)
        removeParentOfOneChild(root,tmp1);
    else if(degree(tmp1)==2)
        removeParentOfTwoChildren(root,tmp1);
}
```

BST.c cont.

```
void removeLeaf (Node** proot, Node* leaf)
{
    Node* pr;
    if(proot == NULL || leaf == NULL)
        return;
    pr = parent(*proot, leaf);
    if (pr==leaf) /*remove root*/
        *proot=NULL;
    else if (pr->left==leaf)
        pr->left = NULL;
    else
        pr->right = NULL;
    free(leaf);
}
```

BST.c cont.

```
void removeParentOfOneChild (Node** root, Node* node)  {
   Node* pr;
   if(root == NULL || node == NULL)
      return;
   pr = parent ( *root, node);
   if (pr==node)  { /*remove root*/
      if (node->left != NULL)
         *root=node->left;
      else
         *root=node->right;
   }
   else if (pr->left==node)  {
      if (node->left != NULL)
         pr->left=node->left;
      else
         pr->left=node->right;
   }
   else {
      if (node->left != NULL)
         pr->right=node->left;
      else
         pr->right=node->right;
   }
   free(node);
}
```

```
:
void removeParentOfTwoChildren(Node** root, Node* node)
{
   if(root == NULL || node == NULL)
      return;

   {
      Node* tmp = maxNode(node->left);
      swap(node, tmp);
      if (degree(tmp)==0)
         removeLeaf (&(*root)->left,tmp);
      else if ( degree(tmp)==1 )
         removeParentOfOneChild (&(*root)->left,tmp);
   }
}
```

BST.c cont.

```
:
Node* parent(Node* root, Node* node)
{
    if(root==NULL || node==NULL)
        return NULL;
    if (root == node)
        return root;
    while (root != NULL)
    {
        if (root->left == node || root->right == node)
            return root;
        else if(node->num  <  root->num)
            root = root->left;
        else
            root = root->right;
    }
    return NULL;
}
```

END