# Data Structures and Algorithms

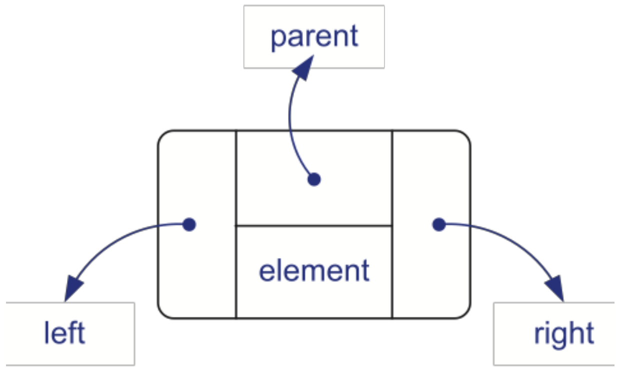
Lecture 21: Binary Trees (cont.)

Nopadon Juneam
Department of Computer Science
Kasetsart university

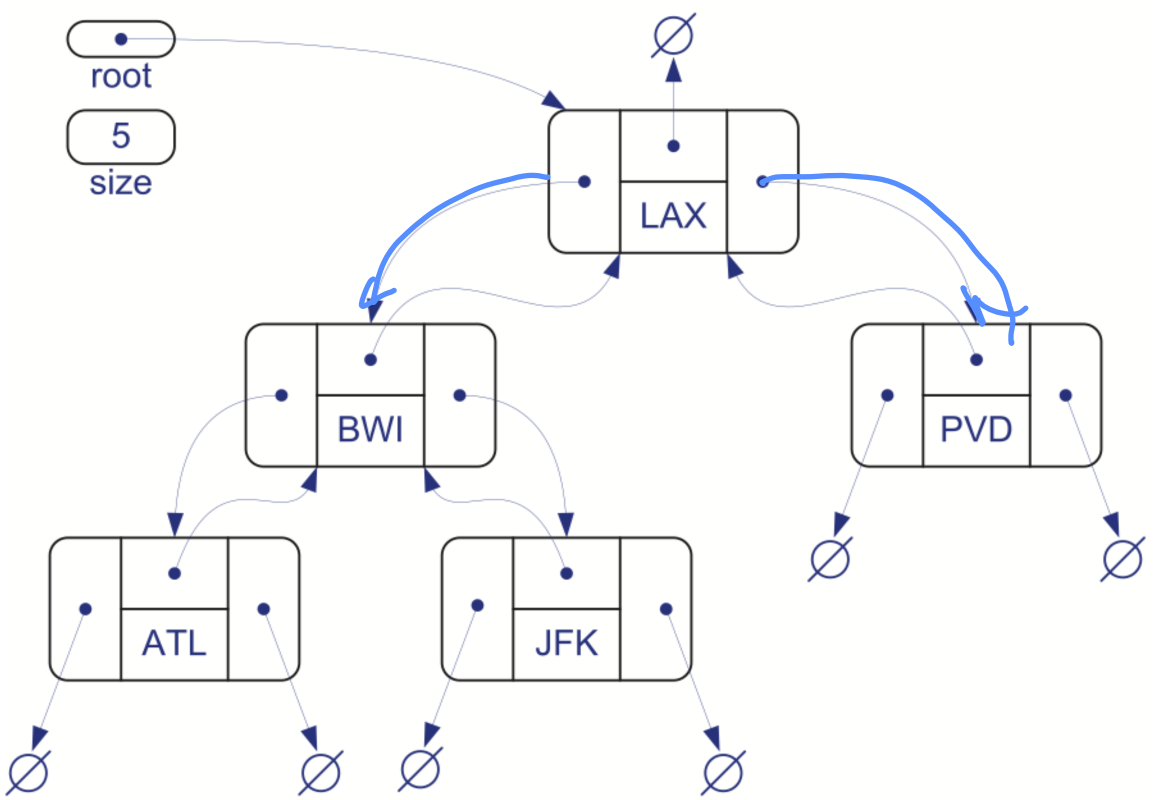
# Outlines

- Data structures for representing binary trees
    - Linked structure
    - Array-based structure
- Operations on binary trees

# Linked Structure for Binary Trees




- In a linked structure for a binary tree *T*, we represent each node of *T* by an object *p* with the following fields:
  - A reference to the node’s element.
  - A link to the node’s parent.
  - A link to the node’s two children.



# Create a Binary Tree (1)

```
#include<stdlib.h>

struct node
{
   int key;
   struct node* parent;
   struct node* left;
   struct node* right;
};

struct node* createNode(int key)
{
  // New node
    struct node* node = (struct node*)malloc(sizeof(struct node));

  // Assign key to this node
  node->key = key;

  // Initialize parent, left child, and right child as NULL
  node->parent = NULL;
  node->left = NULL;
  node->right = NULL;
  return(node);
}
```

# Create a Binary Tree (2)

```
struct node* createLeft(int key, struct node* parent)
{
    // Insert new node as a left child of parent
    struct node* node = createNode(key);
    parent->left = node;
    node->parent = parent;
    return(node);
}


struct node* createRight(int key, struct node* parent)
{
    // Insert new node as a right child of parent
    struct node* node = createNode(key);
    parent->right = node;
    node->parent = parent;
    return(node);
}
```

# Create a Binary Tree (3)

```
int main()
{
  /*create root*/
  struct node *root = createNode(1);
  /* following is the tree after above statement
      1
     /  \
    NULL  NULL
  */
  createLeft(2, root);
  createRight(3, root);
  /* 2 and 3 become children of 1
        1
       /  \
      2     3
    /   \   / \
   NULL NULL NULL NULL
  */
  createLeft(4, root->left);
  /* 4 becomes left child of 2
        1
      /     \
     2        3
    /  \     /  \
   4   NULL  NULL  NULL
  /  \
NULL NULL
*/
  return 0;
}
```

# Operations on Binary Trees (1)

- Basic operations performed on a binary tree *T* includes
  - createNode(*u*, *T*): create a new node *u* to be later inserted into *T*
  - createLeft
  - insertLeft(*u*, *p*, *T*): create a new node *u* as a left child of existing node *p* in *T*
  - createRight
  - insertRight(*u*, *p*, *T*): create a new node *u* as a left child of existing node *p* in *T*
  - getParent(*u*, *T*): return the parent of *u* in *T* S return node -> parent;
  - getLeft(*u*, *T*): return the left child of *u* in *T* return node -> left;
  - getRight(*u*, *T*): return the right child of *u* in *T* return node -> right;

# Operations on Binary Trees (2)

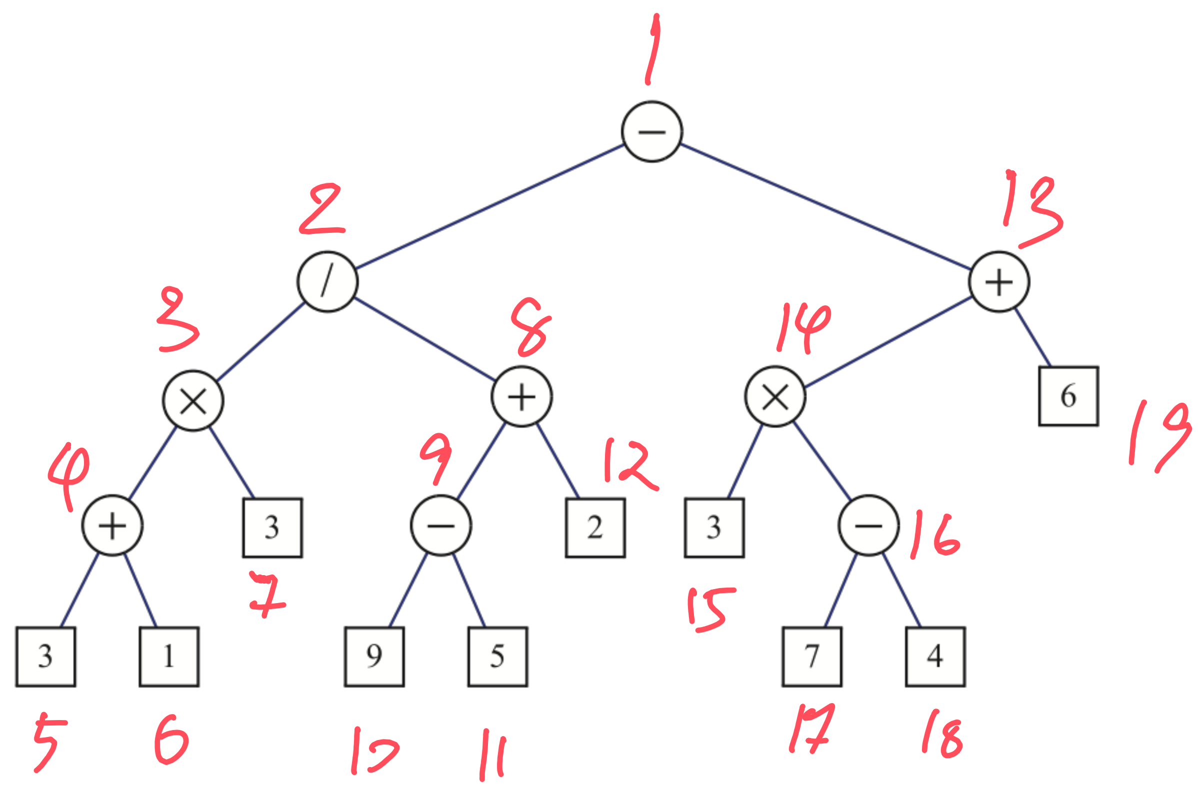
- More basic operations performed on a binary tree *T* includes
  - isRoot(*u*, *T*): check whether a given node *u* is the root of *T* S
  - isExternal(*u*, *T*): check whether a given node *u* is an external node (leaf) of *T*
  - depth(*u*, *T*): return the depth of node *u* in *T* S
  - preorder(*r*, *T*): perform a preorder traversal of *T*, starting with the root *r* of *T*
  - postorder(*r*, *T*): perform a postorder traversal of *T*, starting with the root *r* of *T*
  - inorder(*r*, *T*): perform a postorder traversal of *T*, starting with the root *r* of *T*
  - height(r,*T*): return the height of *T* that is rooted at *r*

# Preorder Traversal: Pseudocode (Root, Left, Right)

- In preorder traversal of a binary tree *T*, we visit the *root* of *T* first and then recursively traverse the *left subtree* and the *right subtree*, respectively

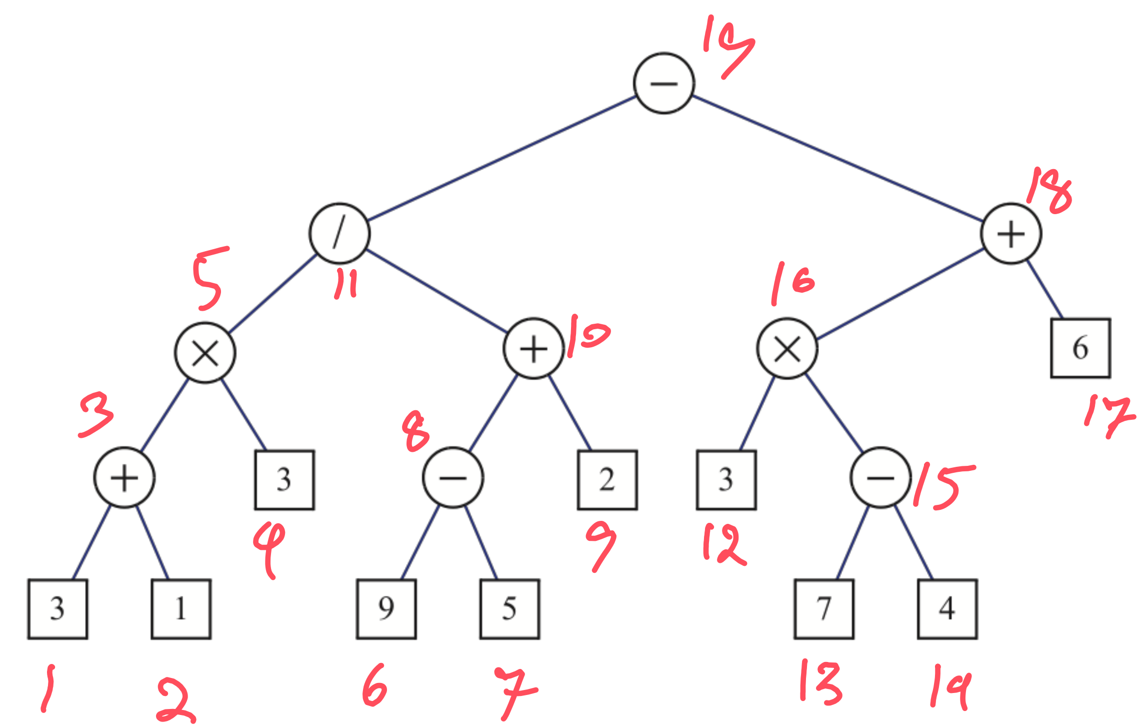
```
preorder(r,T):
    visit node r
    if r.left is not empty:
        preorder(r.left,T)
    if r.right is not empty:
        preorder(r.right,T)
```

Recursively traverse left subtree



# Postorder Traversal: Pseudocode (Left, Right, Root)

- In post traversal of a binary tree *T*, we recursively traverse the *left subtree* and the *right subtree*, respectively, and then visit the *root*



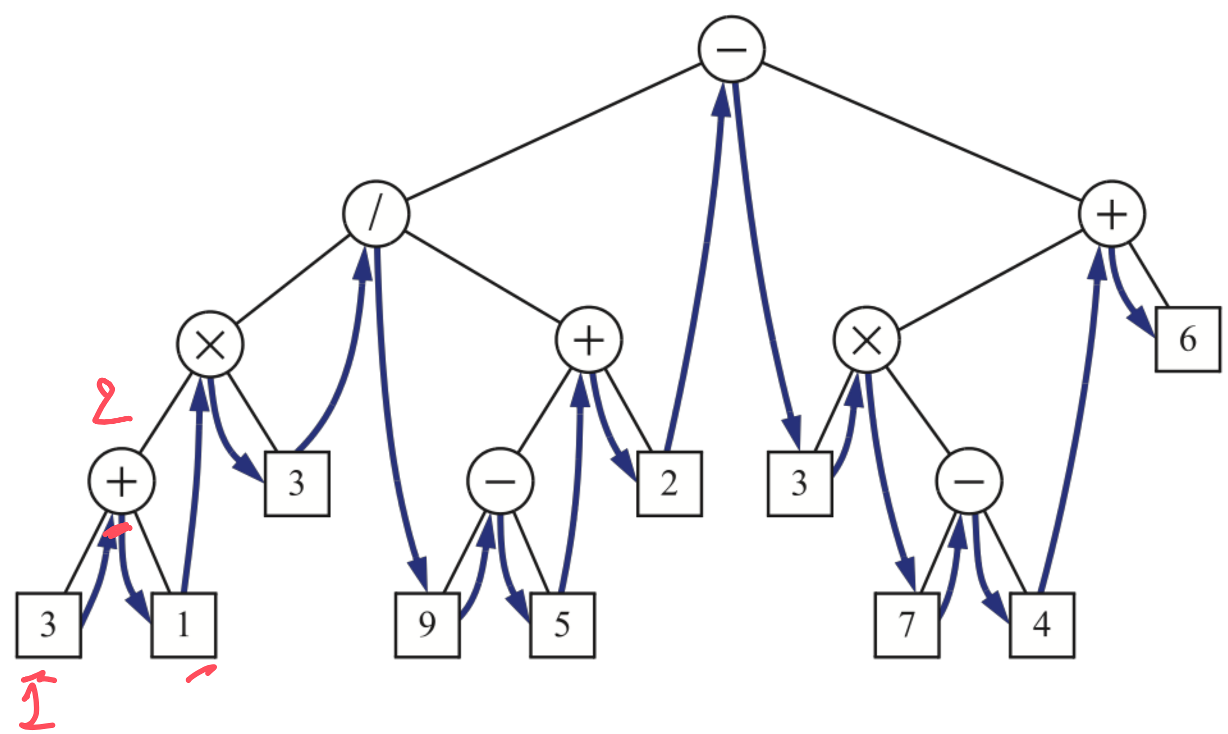
```
postorder(r,T):
    if r.left is not empty:
        postorder(r.left,T)
    if r.giht is not empty:
        postorder(r.right,T)
    visit node r
```

ln

# Inorder Traversal: Pseudocode (Left, Root, Right)

- In inorder traversal of a binary tree *T*, we recursively traverse the *left subtree* first, then visit the *root* of *T*, and finally recursively traverse the *right subtree*

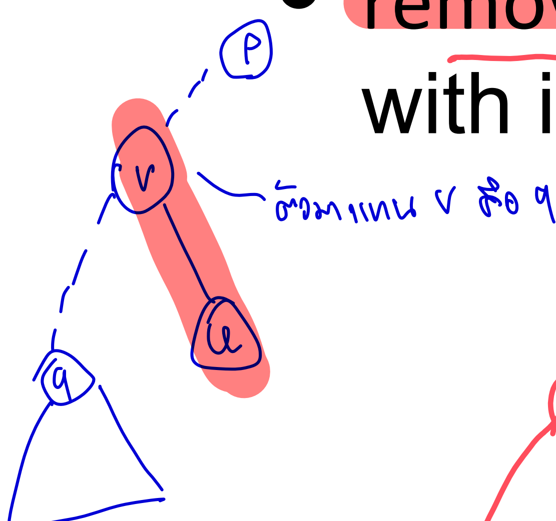
```
inorder(r,T):
    if r.left is not empty:
        inorder(r.left,T)
    visit node r
    if r.right is not empty:
        inorder(r.right,T)
```

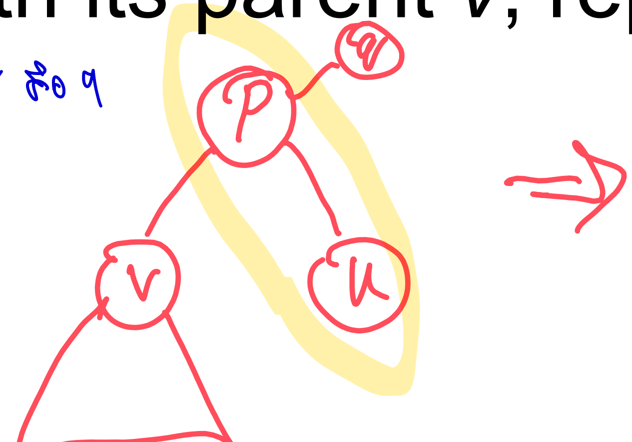


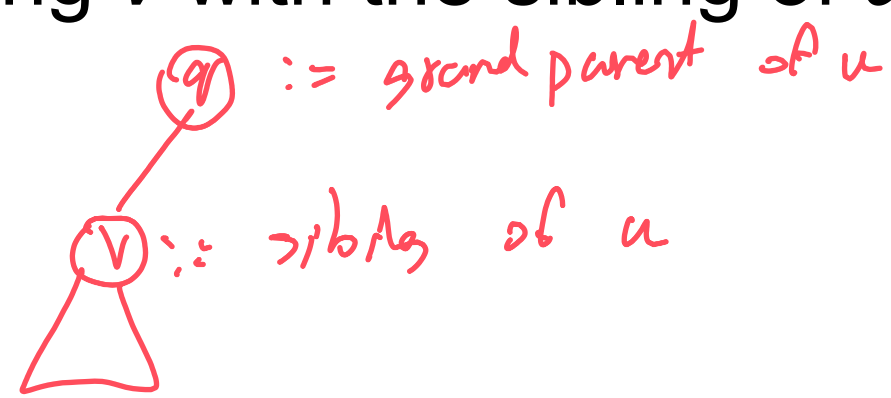
# More Operations on Binary Trees


- expandExternal(*u*, *T*): transform node *u* from being external into internal by creating two new external nodes and making them left and right children of *u*

- removeAboveExternal(*u*, *T*): remove the external node *u* with its parent *v*, replacing *v* with the sibling of *u*



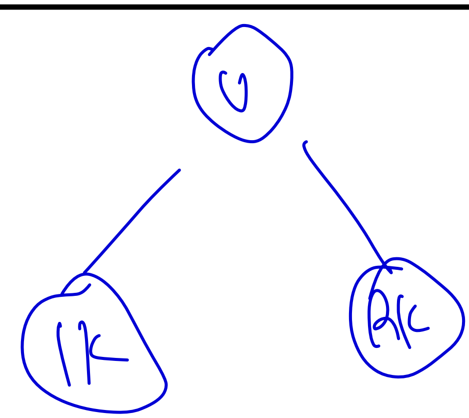



# Operation: expandExternal

- expandExternal(*u*, *T*): transform node *u* from being external into internal by creating two new external nodes and making them left and right children of *u*

u

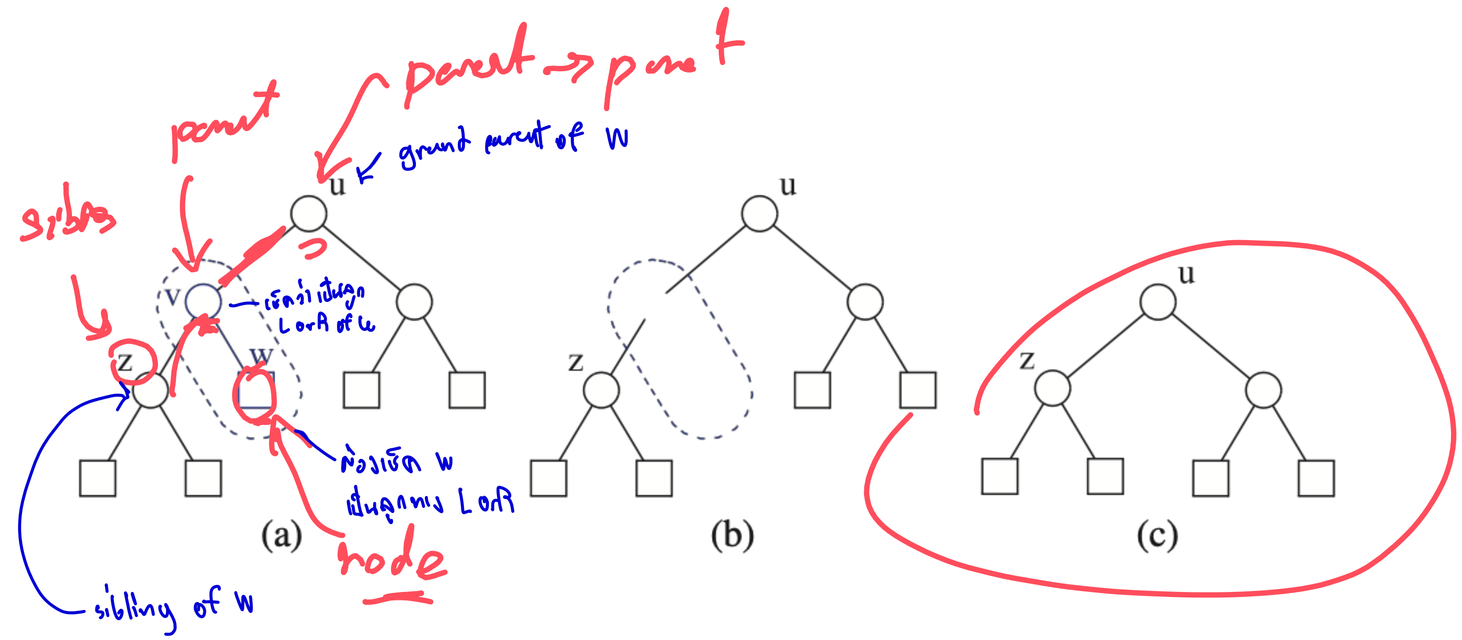
```
void expandExternal(struct node* node, int  leftKey, int rightKey)
{
   createLeft(leftKey, node);
   createLeft(rightKey, node);
}
```



# Operation: removeAboveExternal

- removeAboveExternal(*w*, *T*): remove the external node *w* with its parent *v*, replacing *v* with the sibling of w

```
struct node* removeAboveExternal(struct node* node)
{
   struct node* parent = node->parent;
   struct node* sibling = (node != parent->left) ? parent->left :
parent-> right);
   if(parent->parent == NULL) { เช็คว่ามี Parent ของ parent รึป่าว
      sibling->parent = NULL; sibling จะเป็น root แทน
   }
   else {                          u
      struct node* grandParent = parent->parent;
      if(parent == grandParent->left) v เป็น L ของ u มั้ย
         grandParent->left = sibling;
      else
         grandParent->right = sibling;
      sibling->parent = grandParent;
   }
   free(parent);
   free(node);
   return(sibling);
}
```



# Complexity of Operations on Binary Trees Using Linked Structure

| Operations | Complexity |
|---|---|
| createRoot/createLeft/CreateRight | $O(1)$ |
| getParent | $O(1)$ |
| getLeft/getRight | $O(1)$ |
| isRoot | $O(1)$ |
| isExternal | $O(1)$ |
| depth | $O(n)$,<br>where $n$ is the number of nodes of binary tree |
| preorder/postorder/inorder | $O(n)$ |
| expandExternal | O(1) |
| removeAboveExternal | O(1) |
| height | $O(n)$ modify Postorder |
| space to store  tree | $O(n)$ |

สิ่งการ access ผ่าน index

# Array-Based Structure for Binary Trees (1)

- An ***array-based structure*** for representing a binary tree *T* is based on a way of ***numbering*** the nodes of *T*

หมายเลขของ Node v

- For every node *v* of *T*, let *f*(*v*) be the integer defined as follows:

  v ที่เป็น Root จะ = 1

  - If v is the root of *T*, then *f*(*v*)=1

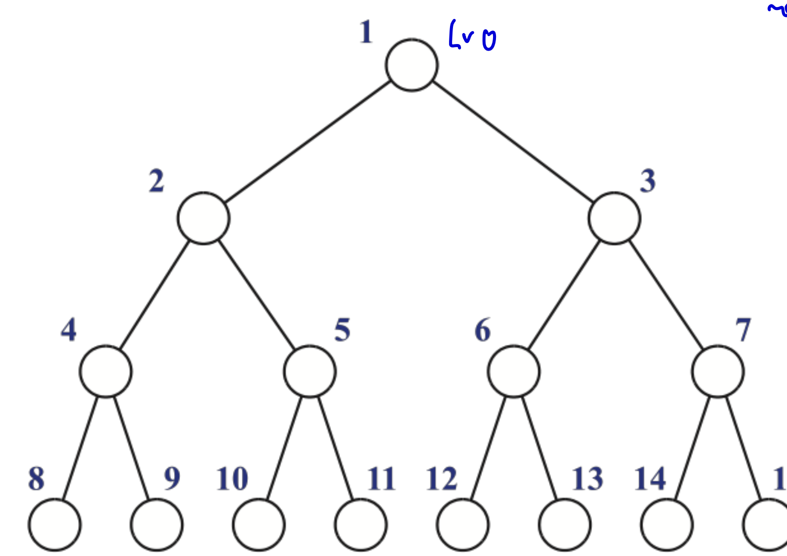
    ถ้า v เป็น L child

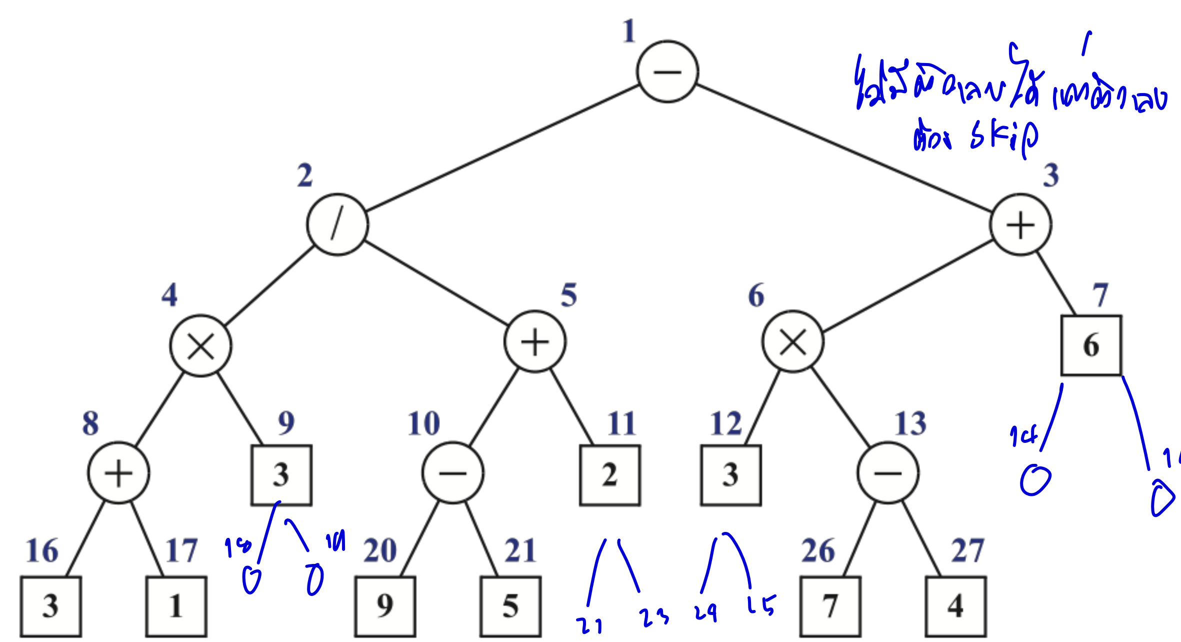
  - If *v* is the left child of node *u*, then *f*(*v*) = 2*f*(*u*)

    ถ้า v เป็น R child

  - If *v* is the right child of node u, then *f*(*v*) = 2*f*(*u*)+1

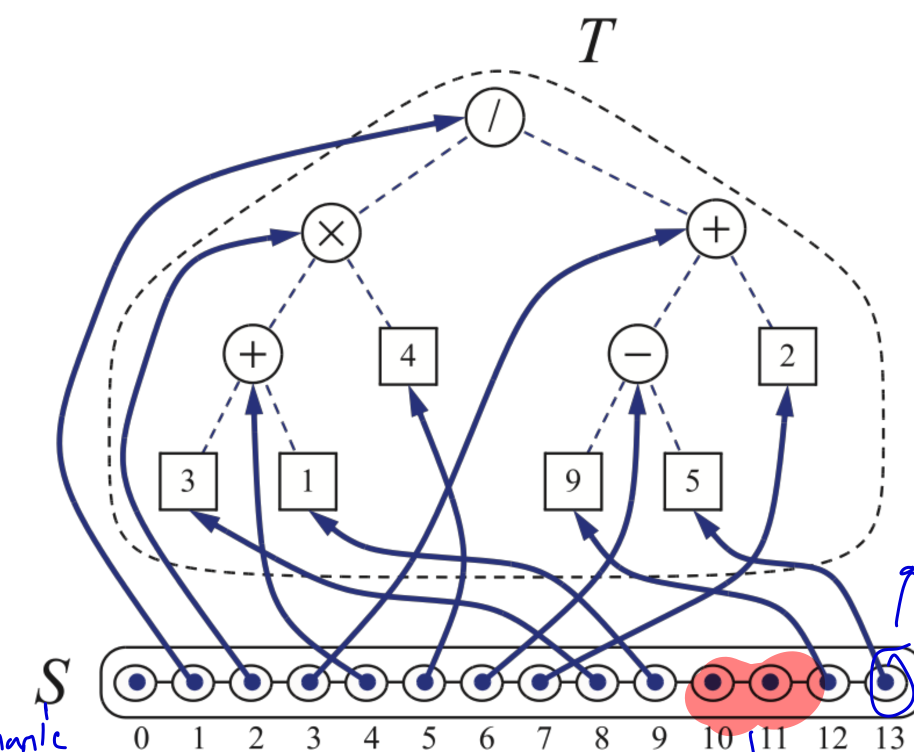
- *Note*: The numbering function *f* is known as a ***level numbering*** of the nodes in a binary tree, because it numbers the nodes on each level of *T* in increasing order from left to right, though it may skip some numbers





# Array-Based Structure for Binary Trees (2)

- The level numbering function *f* suggests a representation of a binary tree *T* by means of a vector *S*, such that node *v* of *T* is associated with the element of *S* at position (index) *f*(*v*)
- Typically, we realize the vector S by means of an extendable array
- Most basic operations can be performed with simple arithmetic operations on the numbering function *f*




ลูก n

Dynamic array

vector

ถูก skip

ยิ่ง ↑ สูง ความแหว่ง จะยิ่งมี waste ใน array

array base access เร็วกว่า linked

# Array-Based Structure for Binary Trees (3)

N จำนวนสมาชิกของ S ที่ต้องทำทั้งหมด

ค่าของ f(v) ที่มากที่สุด

- Let $n$ be the number of nodes of $T$, and let $f_M$ be the maximum value of $f(v)$ over all the nodes of $T$. The vector $S$ has size $N = f_M + 1$, since the element of $S$ at index 0 is not associated with any node of $T$

จะมีเผื่อเลขที่ index 0 ตลอดเพราะไม่ใช้

ทดลองมีค่าสูงสุดคือเท่าไหร่? ต้องจอง array เท่าไหร่

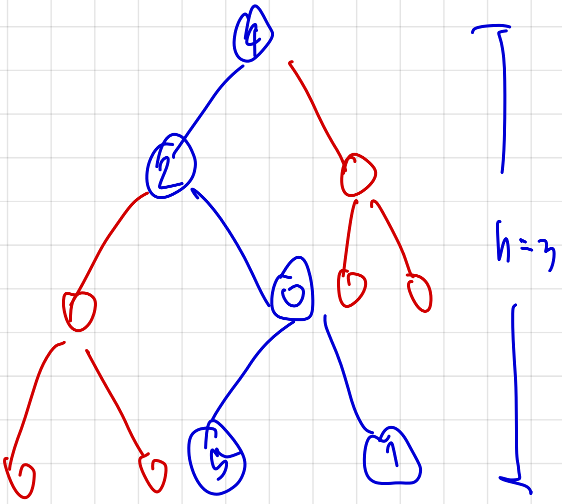
$N \leq 2^{h+1}$

ไม่สามารถที่ใช้การชี้ไปของตัวเลข $n = n+1$

- Additionally, $S$ may have empty elements that do not refer to any existing node of $T$

$N \leq 2^{h+1}$

- For a binary tree of height $h$, $\cancel{N \leq 2^h}$. Since $\log_2(n+1)-1 \leq h \leq n-1$, it follows that $(n+1)/2 \leq N \leq 2^n$. So, at worst, the number of the elements of $S$ grows exponentially in the number of the nodes of $T$




เราจะต้องลอง จำนวนสมาชิกของ S เท่ากับจำนวนของโหนดใน Complete BT ซึ่งมีความสูง h

เริ่มจาก 1, 0 เป็น null

เพราะฉะนั้น $N \leq 2^0 + 2^1 + 2^2 \ldots + 2^h = 2^{h+1} - 1 + 1 = 2^{h+1}$

$\log_2(n+1) - 1 \leq h \leq n-1$

ในกรณีที่ $h = n-1$

$\Rightarrow n \leq 2^{(n-1)+1} = 2^n = O(2^n)$

ในกรณีที่ $h = \log_2(n+1) - 1$

$\Rightarrow N \leq 2^{\log_2(n+1) \cancel{-1+1}}$

$= n+1 = O(n)$

ความสูงของ BT:

- ถ้า $h = O(\log n)$ ถือว่า BT สูงต่ำ (ดีย์!!) $N = O(n)$
- ถ้า $h = O(n)$ ถือว่า BT สูงมาก (ไม่ดีย์!!) $N = O(2^n)$

# Basic Operations on Binary Trees Based on Array-Based Structure

- Examples of basic operations performed on a binary tree *T*:

ตำแหน่งของ Parent ตาม level numbering

ควรรู้ size ของ array ล่วงหน้า

  - insertLeftKey($k$, $\cancel{r}$ i, $T$): S[2*$\cancel{f(r)}$ i] = k

ตำแหน่งของ child

2

4

i = 4, Parent?

parent = 4/2 = 2 ✓ parent

  - getParentKey($\cancel{u}$ i, $T$): return $S[\cancel{f(u)}$ i$/2]$
  - getLeftKey($\cancel{u}$ i, $T$): return $S[2*\cancel{f(u)}$ i$]$
  - isRoot($\cancel{u}$ i): return ($\cancel{f(u)}$ i == 1)
  - isExternal($\cancel{u}$ i): return
    !((~~getLeft~~ Key($\cancel{u}$ i) != NULL) || (~~getRight~~ Key($\cancel{u}$ i) != NULL))
  - depth($\cancel{u}$ i): return floor($\log_2$($\cancel{f(u)}$ i))

# Complexity of Operations on Binary Trees Using Array-Based Structure

| Operations | Complexity |
| --- | --- |
| insertLeftKey/insertRightKey | $O(1)$ |
| getParentKey | $O(1)$ |
| getLeftKey/getRightKey | $O(1)$ |
| isRoot | $O(1)$ |
| isExternal | $O(1)$ |
| depth** | $O(1)$ |
| preorder/postorder/inorder | $O(n)$<br>where $n$ is the number of nodes of binary tree |
| expandExternal | $O(2^h)$<br>where $h$ is the height of binary tree |
| removeAboveExternal | $\cancel{O(1)}$ $O(2^h)$ |
| height | $O(n)$ |
| space to store tree | $O(2^h)$ |