Data Structures

Lecture 18: Binary Trees (cont.)

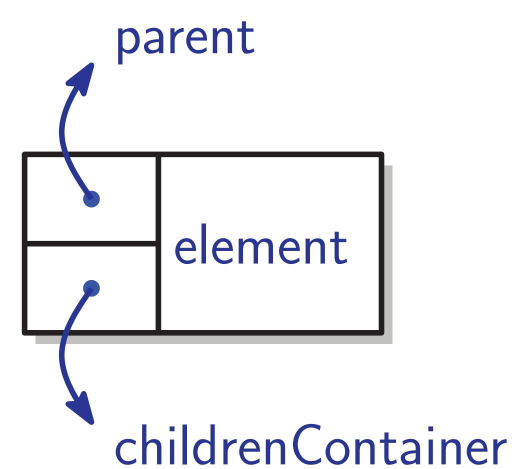
Nopadon Juneam
Department of Computer Science
Kasetsart university

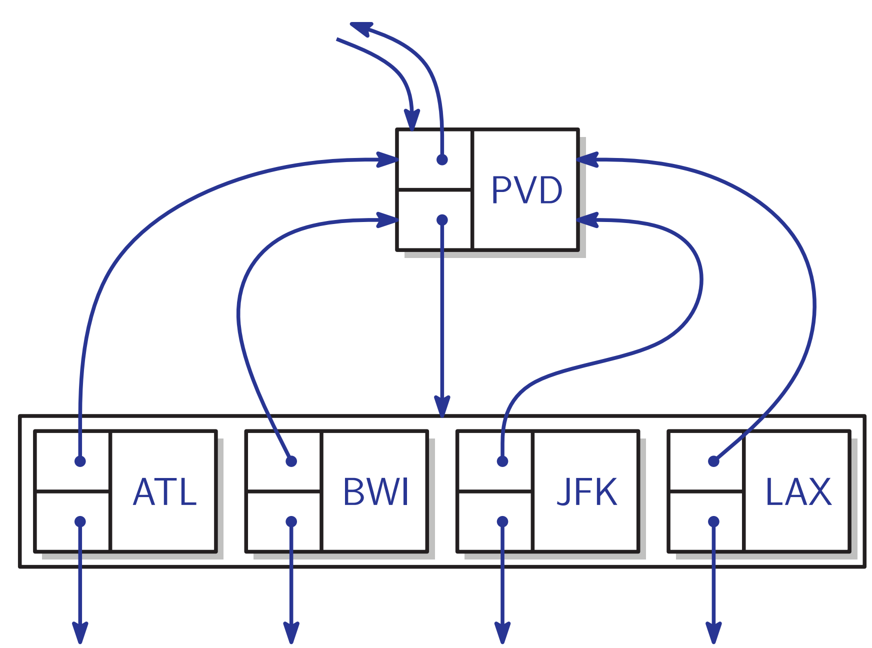
Outlines

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

Linked Structure for General Trees

- A natural way to realize a tree T is to use a ***linked structure***, where 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.
  - Some kind of collection (for example, a list or array) to store links to the node's children.





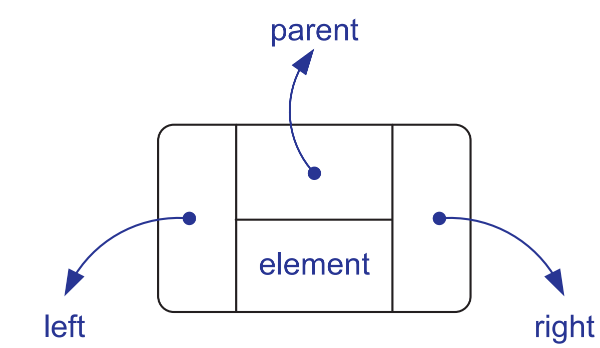
Linked Structure for Binary Trees

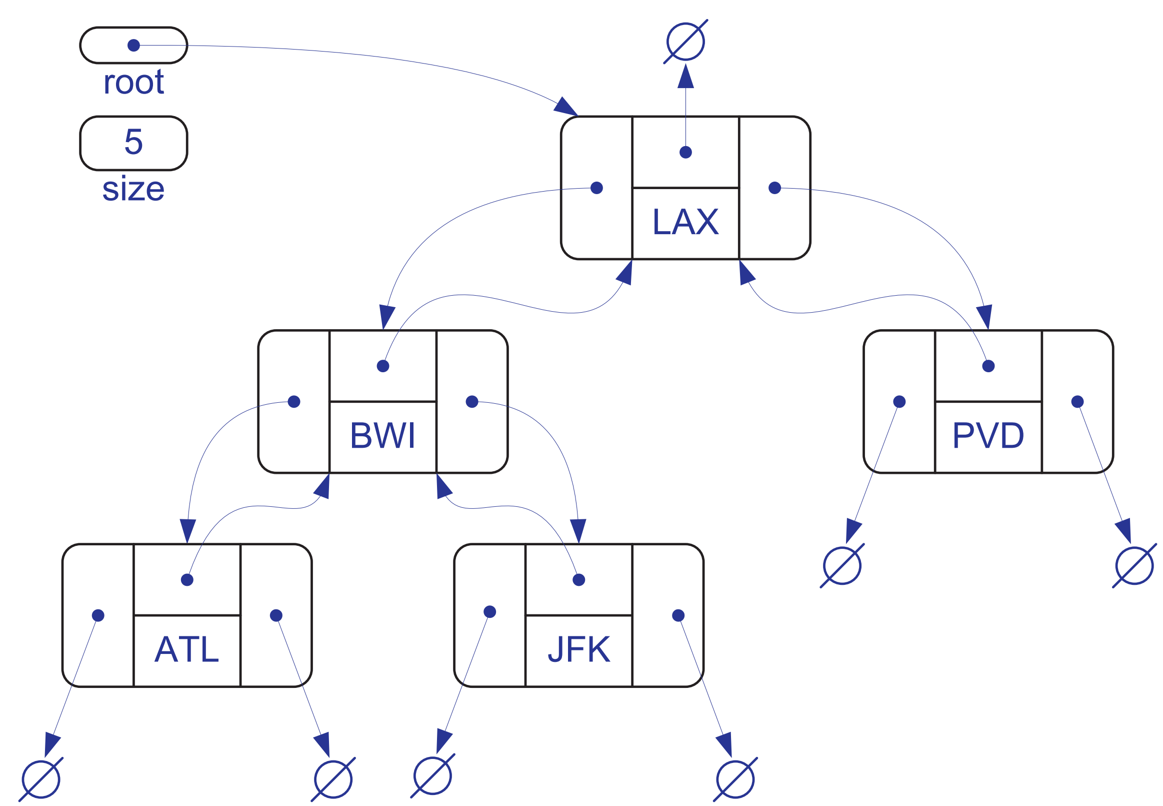
- In a linked structure for a binary tree T, we represent each node of T by a node object p with the following fields:

  - 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)

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

Basic Operations on Binary Trees

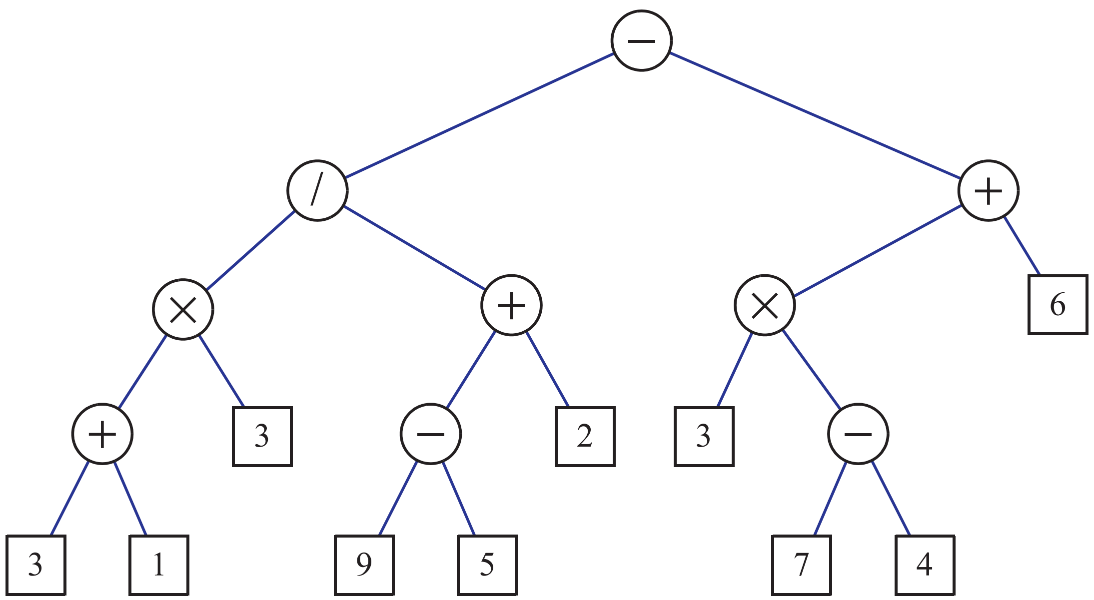
- Basic operations commonly performed a binary tree *T*:
  - `createNode(u, T):` create a node u to be *inserted* in the tree *T*
  - get`Parent(u, T):` return the parent of *u* in *T*
  - `getLeft(u):` return the left child of *u* in *T*
  - `getRight(u):` return the right child of *u* in *T*
  - `isRoot(u, T):` check whether a given node *u* is the root of *T*
  - `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*

More Operations on Binary Trees

- `preorder(r, T):` perform a preorder traversal of *T*
- `postorder(r, T):` perform a postorder traversal of *T*

Preorder Traversal: Pseudocode (Root, Left, Right)

- In a 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

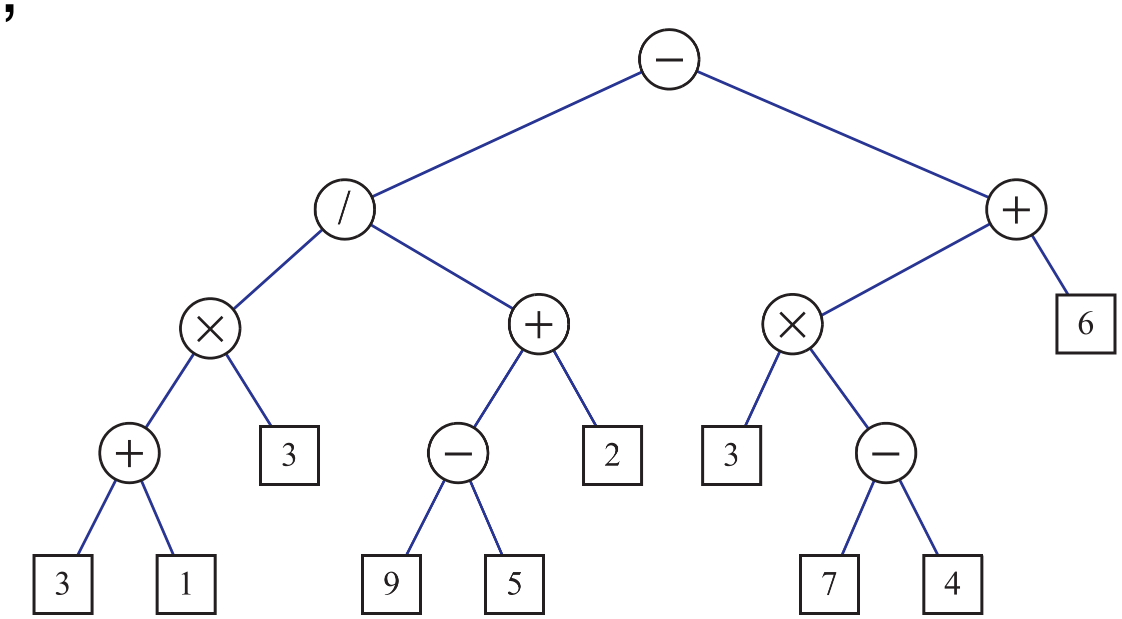



```
preorder(r,T):
   visit node r
   if r is internal:
      preorder(r.left,T)
      preorder(r.right,T)
```

Postorder Traversal: Pseudocode (Left, Right, Root)

- In a 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 is internal:
     postorder(r.left,T)
     postorder(r.right,T)
  visit node r
```



More Operations on Binary Trees Using a Linked Structure

- `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*

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

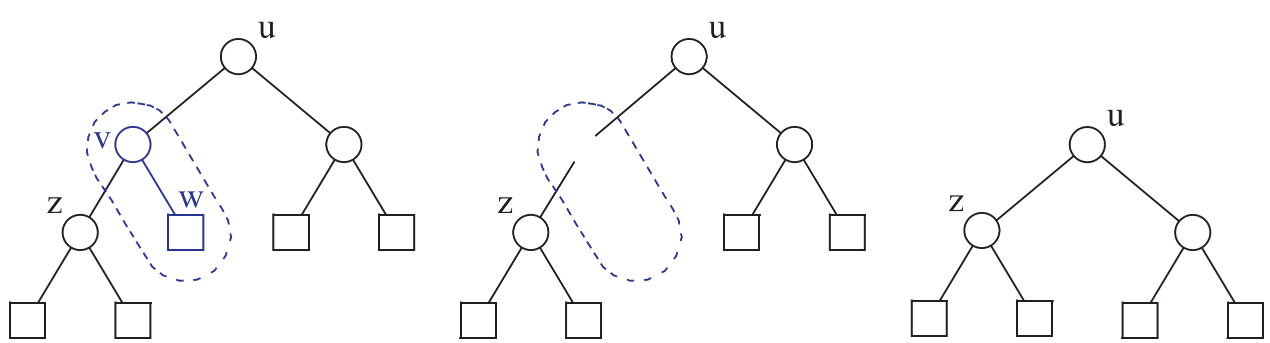
```c
void expandExternal(struct node* node)
{
    struct node* left = createNode();
    node->left = left;
    left->parent = node;
    struct node* right = createNode();
    node->right = right;
    right->parent = node;
}
```

```c
struct node* createNode()
{
    struct node* node = (struct node*)
malloc(sizeof(struct node));
    node->key = 0;
    node->parent = NULL;
    node->left = NULL;
    node->right = NULL;
    return(node);
}
```

```c
int main()
{
    struct node* node1 = createNode();
    node1->key = 1;
    expandExternal(node1);
    node1->left->key = 2;
    node1->right->key = 3;
    expandExternal(node1->left);
    node1->left->left->key = 4;
    node1->left->right->key = 5;
    preorder(node1);
    return 0;
}
```

Operation: removeAboveExternal

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



(a) (b) (c)

```c
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) {
        sibling->parent = NULL;
    }
    else {
        struct node* grandParent = parent->parent;
        if(parent == grandParent->left)
            grandParent->left = sibling;
        else
            grandParent->right = sibling;
        sibling->parent = grandParent;
    }
    free(parent);
    free(node);
    return(sibling);
}
```

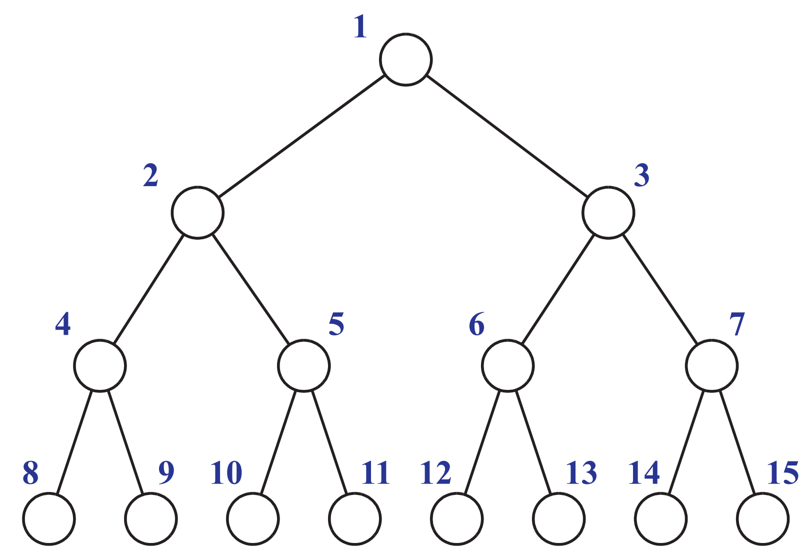
```c
int main()
{
    struct node* node1 = createNode();
    node1->key = 1;
    expandExternal(node1);
    node1->left->key = 2;
    node1->right->key = 3;
    expandExternal(node1->left);
    node1->left->left->key = 4;
    node1->left->right->key = 5;
    preorder(node1);
    printf("\n");
    removeAboveExternal(node1->left->right);
    preorder(node1);
    return 0;
}
```

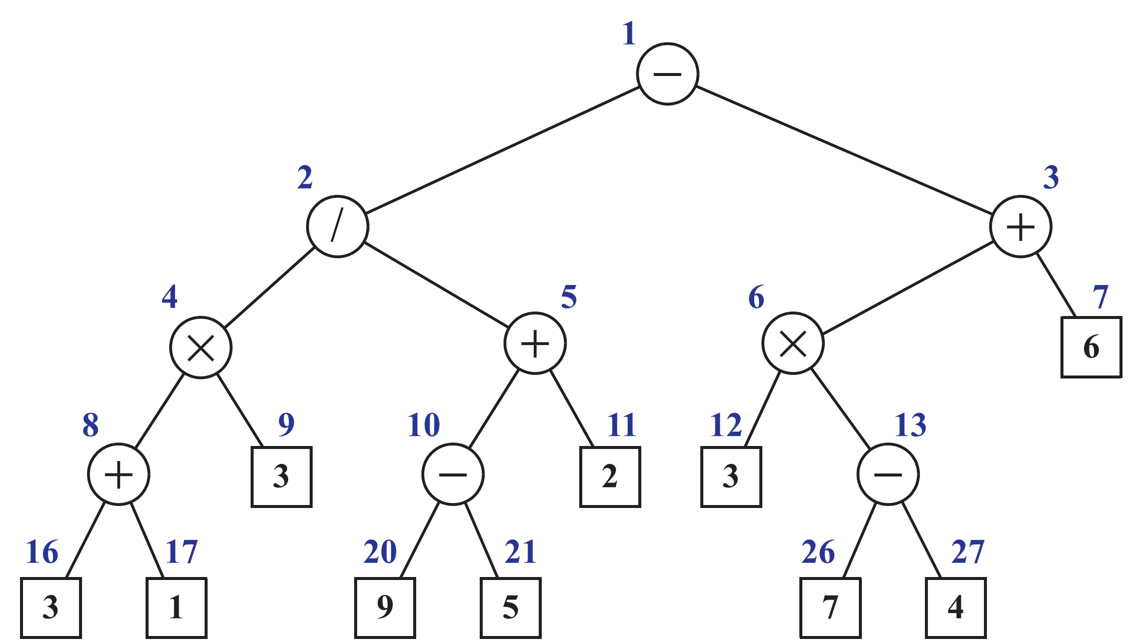
Complexity of Operations on Binary Trees Using a Linked Structure

Operations	Complexity
`createNode`	$O(1)$
`getParent`	$O(1)$
`getLeft`	$O(1)$
`getRight`	$O(1)$
`isRoot`	$O(1)$
`isExternal`	$O(1)$
`depth`	$O(n)$
`preorder`	$O(n)$
`postorder`	$O(n)$
expandExternal	$O(1)$
removeAboveExternal	$O(1)$

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*
- For every node *v* of *T*, let *f*(*v*) be the integer defined as follows:
    - If v is the root of *T*, then *f*(*v*)=1
    - If *v* is the left child of node *u*, then *f*(*v*) = 2*f*(*u*)
    - 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



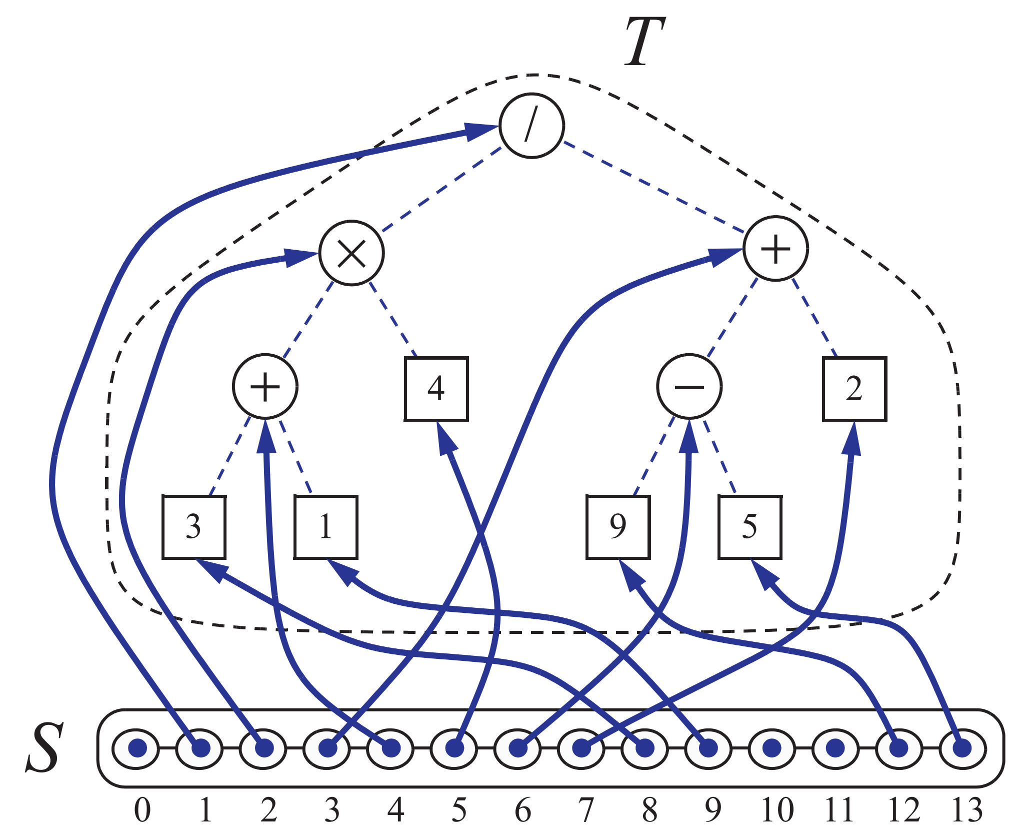


Array-Based Structure for Binary Trees (2)

- 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 *f*(*v*)

- Typically, we realize the vector S by means of an extendable array

- We can use it to easily perform basic operations by using simple arithmetic operations on the numbers *f*(*v*) associated with each node *v* involved in the operations



Array-Based Structure for Binary Trees (3)

- 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. Also, S will have, in general, a number of empty elements that do not refer to existing nodes of T
- For a tree of height h, N = 2^h. In the worst case, this can be as high as $2^n - 1$

Basic Operations on Binary Trees Using an Array-Based Structure

- Basic operations commonly performed a binary tree *T*:
    - `getLeft(i): return 2*i;`
    - `getRight(i): return 2*i+1;`
    - `isRoot(i): return (i == 1);`
    - `isExternal(i): return !((getLeft(i) != NULL) || (getRight(i) != NULL));`
    - `depth(i): return floor(log(i));`

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

Operations	Complexity
`getParent`	$O(1)$
`getLeft`	$O(1)$
`getRight`	$O(1)$
`isRoot`	$O(1)$
`isExternal`	$O(1)$
`depth`	$O(1)$ **
`preorder`	$O(n)$
`postorder`	$O(n)$
expandExternal	$O(2^h)$
removeAboveExternal	$O(n)$

Inorder Traversal: Pseudocode (Left, Root, Right)

- In an 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 has left:
      inorder(r.left,T)
   visit node r
   if r has right:
      inorder(r.right,T)
```