Data Structures and Algorithms

Lecture 18: Trees (cont.)

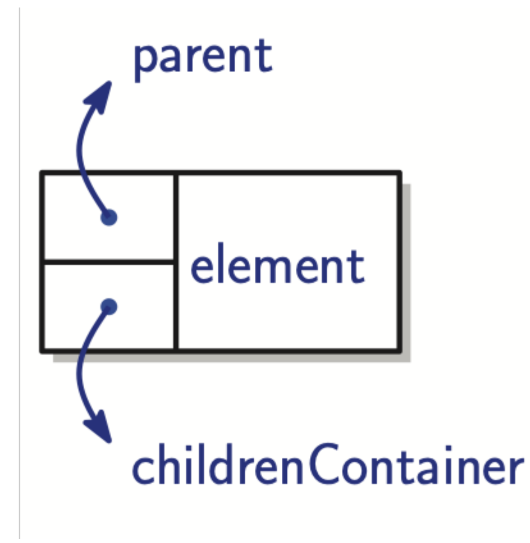
Nopadon Juneam
Department of Computer Science
Kasetsart university

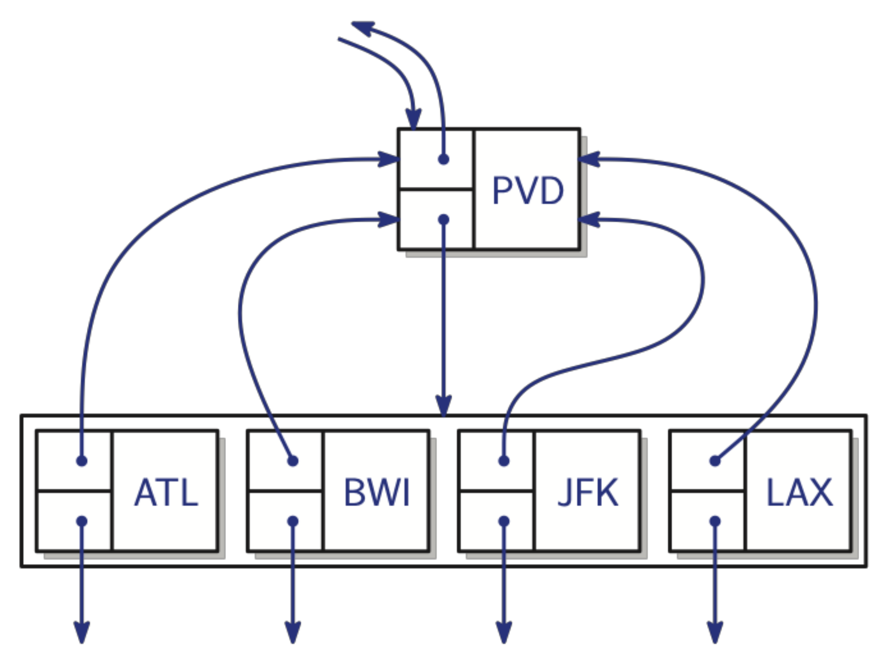
Outlines

- Ordered trees & linked structures
- Basic operations on ordered trees

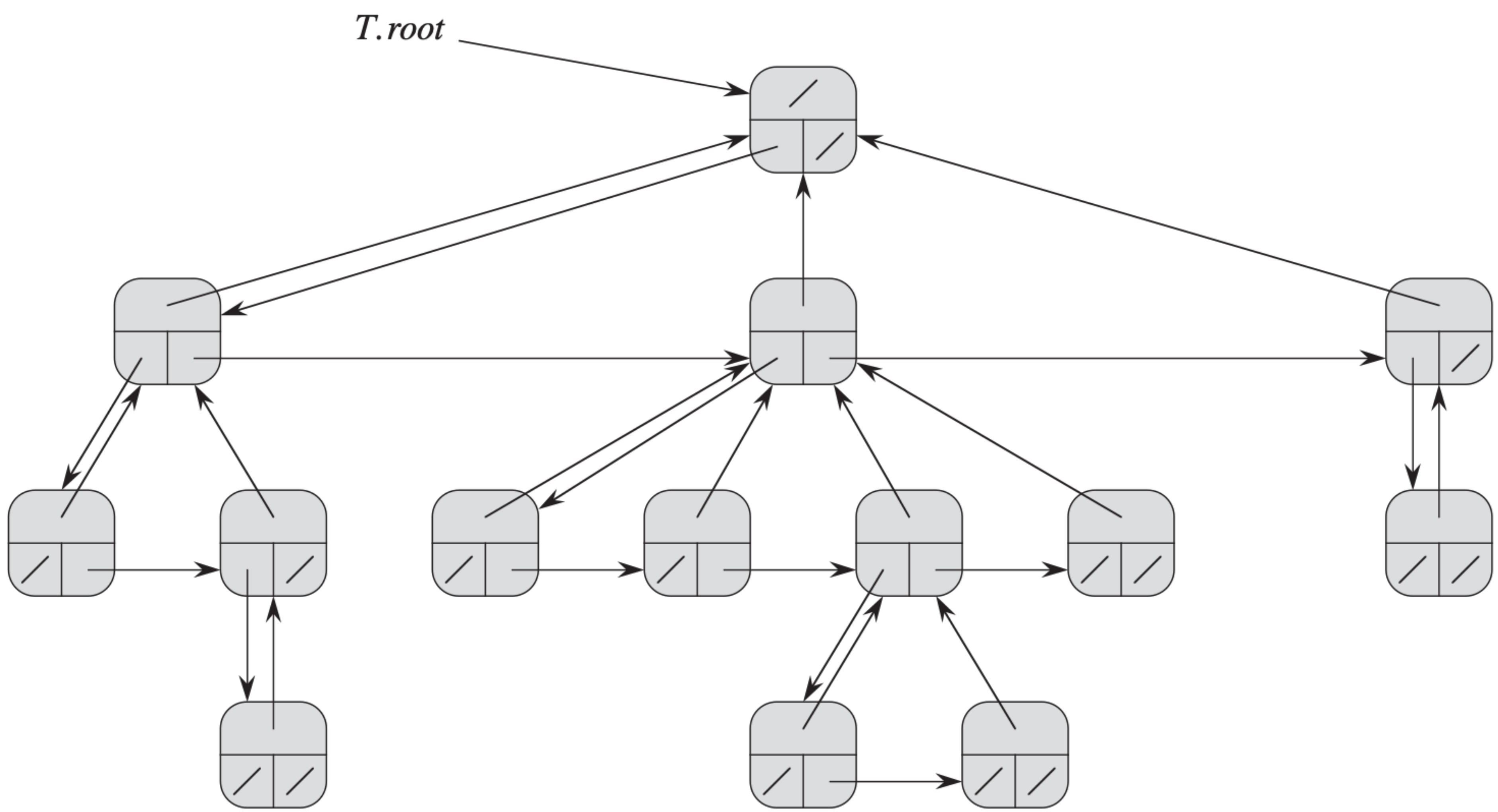
Linked Structure for General Trees

- 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

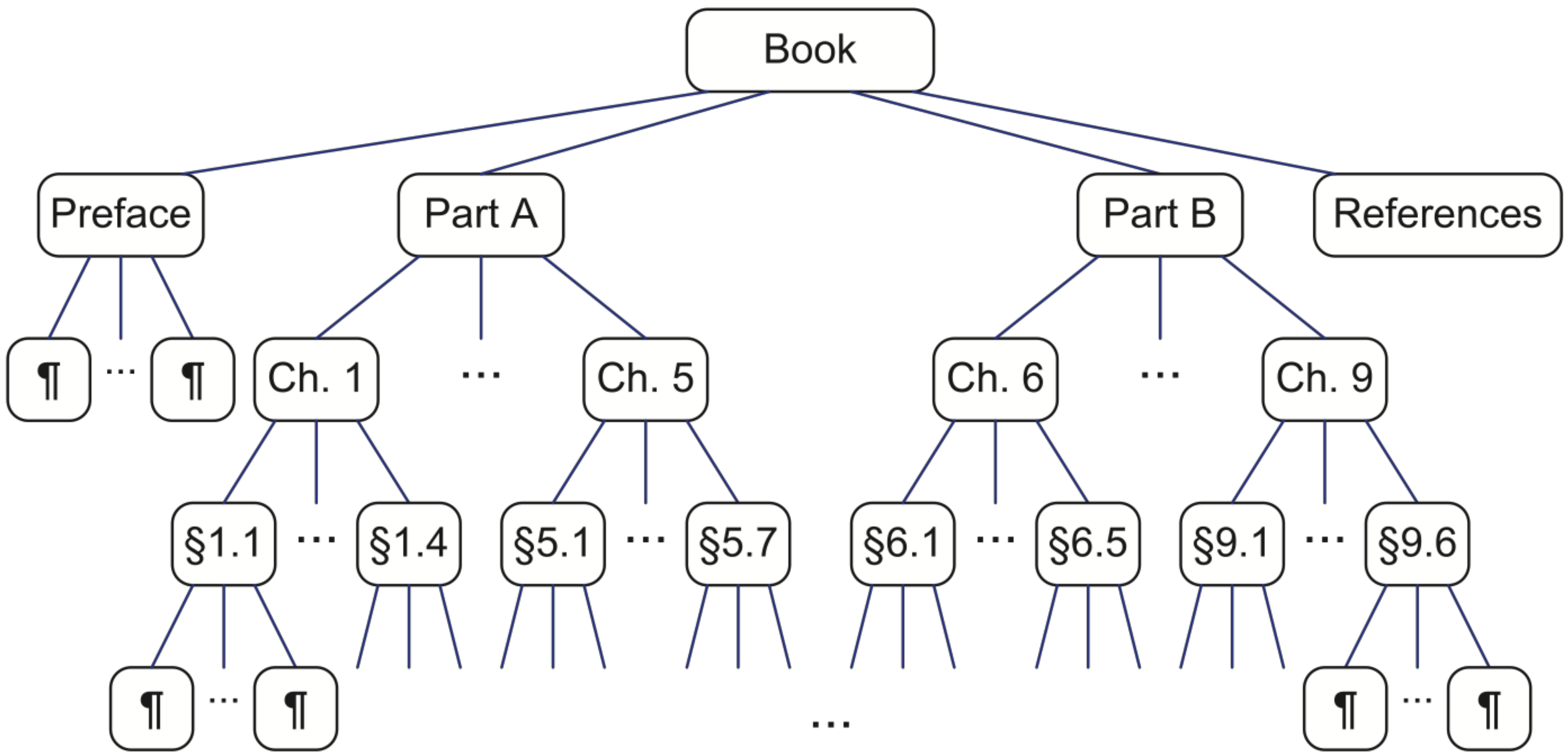




Linked Structure for Rooted Trees



Ordered Trees



- An ***ordered tree*** is a rooted tree in which the children of each node are ordered. That is, if a node has *k* children, then there is a first child, a second child, . . . , and a *k*-th child

Basic Operations on Ordered Trees (1)

- Basic operations commonly performed on an ordered tree:
  - createRoot(*r*, *T*): create the root *r* of a tree *T*
  - createNode(*u*, *p*, *T*): create a node *u* whose parent is node *p* in the tree *T*
  - return u->parent
  - getParent(*u*, *T*): return the parent of node *u* in *T*
  - Return ลูกคนที่เท่าไร
  - getChild(*u*, *k*, *T*): return the *k*-th child of *u* in *T*

Basic Operations on Ordered Trees (2)

- 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
- height(T): return the height of T
  (will talk about it in Tree Traversals)

Operation: createRoot

- createRoot(*r*, *T*): create the root *r* of the tree *T*

```
struct node* createRoot(int key)
{
  // Allocate memory for new node
  struct node* node = (struct node*)
              malloc(sizeof(struct node));
  // Assign key to this node
  node->key = key;
  // Initialize parent
  node->parent = NULL;
  // Initialize left child, and right sibling as NULL

  node->leftChild = NULL;
  node->rightSibling = NULL;
  return(node);
}
```

O(1)

```
struct node
{
   int key;
   struct node* parent;
   struct node* leftChild;
   struct node* rightSibling;
};
```

```
int main()
{
  /*create root*/
  struct node* node1 = createRoot(1);
  ...
    free(node1);
  return 0;
}
```

Operation: createNode

- createNode(*u*, *p*, *T*): create a node u whose parent is node *p* in the tree *T*

```
struct node* createNode(int key, struct node* parent)
{
 // Allocate memory for new node
 struct node* node = (struct node*)malloc(sizeof(struct node));
 // Assign key to this node
 node->key = key;
 // Initialize parent
 node->parent = parent;
 // Initialize left child, and right sibling as NULL
 node->leftChild = NULL;
 node->rightSibling = NULL;
 // Set this node as a child to its parent
 if(node->parent != NULL) {
     if(node->parent->leftChild != NULL) {
        struct node* child = node->parent->leftChild;
        while(child->rightSibling != NULL) {
           child = child->rightSibling;
        }
        child->rightSibling = node;
     }
     else {
        node->parent->leftChild = node;
     }
   }
 return node;
}
```

O(k)

```
struct node
{
   int key;
   struct node* parent;
   struct node* leftChild;
   struct node* rightSibling;
};
```

```
int main()
{
 /*create root*/
 struct node* node1 = createRoot(1);
 struct node* node2 =createNode(2, node1);
 struct node* node3 =createNode(3, node1);
 struct node* node4 =createNode(4, node2);
 /* 4 becomes left child of 2
        1
      /    \
     2      3
    /
   4
 */
    …
 return 0;
}
```

Operations: getParent

- getParent(*u*, *T*): return the parent of *u* in *T*

```
struct node
{
   int key;
   struct node* parent;
   struct node* leftChild;
   struct node* rightSibling;
};
```

O(1)

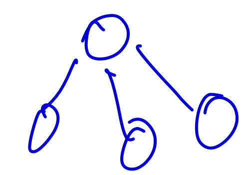
```
struct node* getParent(struct node* node)
{
   return node->parent;
}
```

```
int main()
{
  /*create root*/
  struct node* node1 = createRoot(1);
  struct node* node2 =createNode(2, node1);
  struct node* node3 =createNode(3, node1);
  struct node* node4 =createNode(4, node2);
  /* 4 becomes left child of 2
         1
      /     \
     2       3
    /
   4
  */
 printf("%d\n", getParent(node2)->key); // output "1"
  ...
 return 0;
}
```

Operations: getChild

- getChild(*u*, *k*, *T*): return the *k*-th child of *u* in *T*

k = degree of u



n-1 node

```
struct node
{
    int key;
    struct node* parent;
    struct node* leftChild;
    struct node* rightSibling;
};
```

O(k)

```
struct node* getChild(struct node* node, int k)
{
    struct node* child = node->leftChild;
    for(int i=1; i<k; i++) {
        child = child->rightSibling;
    }
    return child;
}
```

```
int main()
{
  /*create root*/
  struct node* node1 = createRoot(1);
  struct node* node2 =createNode(2, node1);
  struct node* node3 =createNode(3, node1);
  struct node* node4 =createNode(4, node2);
  /* 4 becomes left child of 2
          1
        /   \
       2     3
      /
     4
  */
 printf("%d\n", getChild(node1,2)->key); // output “3”
 ...
 return 0;
}
```

Operation: isRoot

- isRoot(*u*, *T*): check whether a given node *u* is the root of *T*

```
struct node
{
   int key;
   struct node* parent;
   struct node* leftChild;
   struct node* rightSibling;
};
```

O(1)

```
void isRoot(struct node* node)
{
   if(node->parent == NULL)
      printf("Yes\n");
   else
      printf("No\n");
}
```

```
int main()
{
 /*create root*/
 struct node* node1 = createRoot(1);
 struct node* node2 =createNode(2, node1);
 struct node* node3 =createNode(3, node1);
 struct node* node4 =createNode(4, node2);
 /* 4 becomes left child of 2
        1
      /   \
     2     3
    /
   4
 */
 isRoot(node1); // output "Yes"
 isRoot(node2); // output "No"
 ...
 return 0;
}
```

Operation: isExternal (isInternal)

- isExternal(*u*, *T*): check whether a given node *u* is an external node (leaf) of *T*

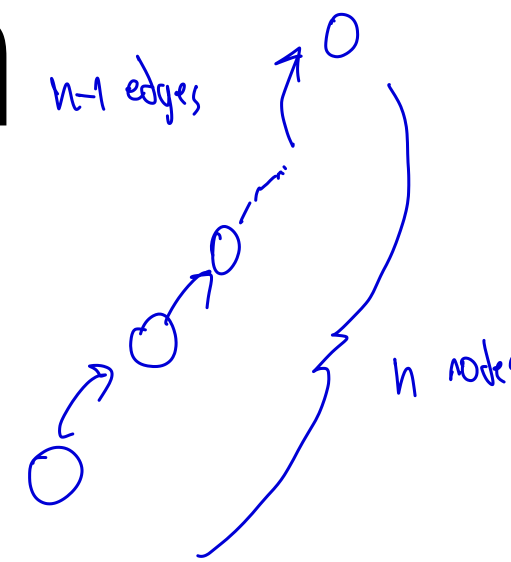
```
struct node
{
   int key;
   struct node* parent;
   struct node* leftChild;
   struct node* rightSibling;
};
```

O(1)

```
void isExternal(struct node* node)
{
   if(node->leftChild == NULL)
      printf("Yes\n");
   else
      printf("No\n");
}
```

```
int main()
{
  /*create root*/
  struct node* node1 = createRoot(1);
  struct node* node2 =createNode(2, node1);
  struct node* node3 =createNode(3, node1);
  struct node* node4 =createNode(4, node2);
  /* 4 becomes left child of 2
        1
      /   \
     2      3
    /
   4
  */
 isExternal(node2); // output "No"
 isExternal(node3); // output "Yes"
 ...
 return 0;
}
```

Operation: depth



- depth(*u*, *T*): return the depth of node *u* in *T*

```
struct node
{
   int key;
   struct node* parent;
   struct node* leftChild;
   struct node* rightSibling;
};
```

```
int depth(struct node* node)
{
   int depth = 0;
   while(node->parent != NULL) {
      node = node->parent;
      depth++;
   }
   return depth;
}
```

O(n)

```
int main()
{
  /*create root*/
  struct node* node1 = createRoot(1);
  struct node* node2 =createNode(2, node1);
  struct node* node3 =createNode(3, node1);
  struct node* node4 =createNode(4, node2);
  /* 4 becomes left child of 2
        1
      /   \
     2     3
    /
   4
  */
 printf("%d\n", depth(node4)); // output "2"
   ...
 return 0;
}
```

Complexity of Operations on Ordered Trees

Operations	Complexity
createRoot	$O(1)$
createNode	$O(k)$, where k is the maximum number of children nodes that a parent node may have in a tree
getParent	$O(1)$
getChild	$O(k)$
isRoot	$O(1)$
isExternal	$O(1)$
depth	$O(n)$, where n is the number of nodes of a tree
height	$O(n)$
space to store tree	$O(n)$