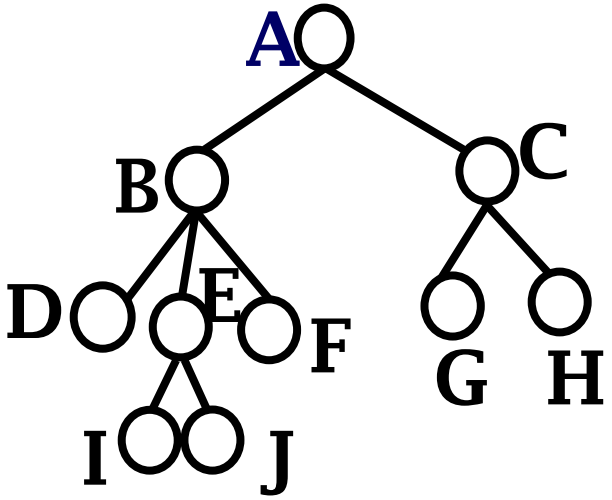
Data Structures and Algorithms (6)

Instructor: Ming Zhang
Textbook Authors: Ming Zhang, Tengjiao Wang and Haiyan Zhao
Higher Education Press, 2008.6 (the "Eleventh Five-Year" national planning textbook)

https://courses.edx.org/courses/PekingX/04830050x/2T2014/

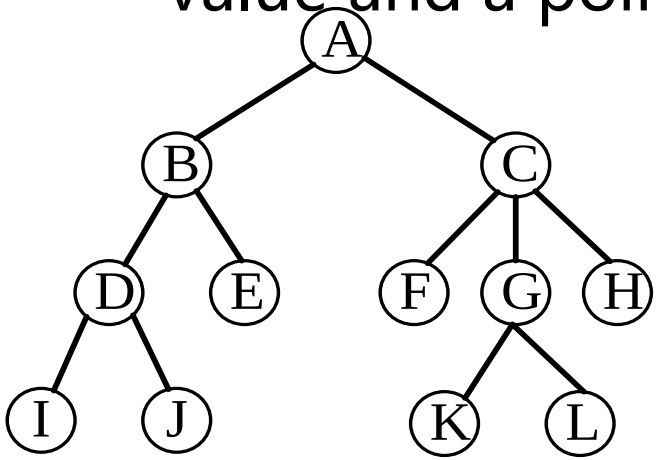
Chapter 6 Trees



- General Definitions and Terminology of Tree
- Linked Storage Structure of Tree
  - List of Children
  - Static Left-Child/Right-Sibling representation
  - Dynamic representation
  - Dynamic Left-Child/Right-Sibling representation
  - Parent Pointer representation and its Application in Union/Find Sets
- Sequential Storage Structure of Tree
- K-ary Trees

Parent Pointer representation

- A representation that you only need to know the parent node
- For each node, you only need to store a pointer which points to its parent node, so that we call it parent pointer representation
- Use an array to store the tree nodes, and each node includes a value and a pointer which points to its parent node



Node index	0	1	2	3	4	5	6	7	8	9	10	11
Value	A	B	C	D	E	F	G	H	I	J	K	L
Parent node index		0	0	1	1	2	2	2	3	3	6	6

Parent Pointer representation: Algorithm

- Find the root node of the current node
  - Start from a node, find a path from that node to its root node
    - O(k), k is the height of the tree
- Check whether two nodes are in the same tree
  - If these two nodes have the same root node, then they are sure to be in the same tree.
  - If these two nodes have different root nodes, then they are sure to be in different trees.

Union/Find Sets

- **Union/Find Sets** is a special kind of sets, consisted of some disjoint subsets. The basic operations of Union/Find Sets are:
  - Find: Find the set the node belongs to
  - Union: Merge two sets
- Union/Find Sets is an important abstract data types
  - The application of Union/Find sets is mainly to solve the problem of equivalence classes .

Equivalence relation

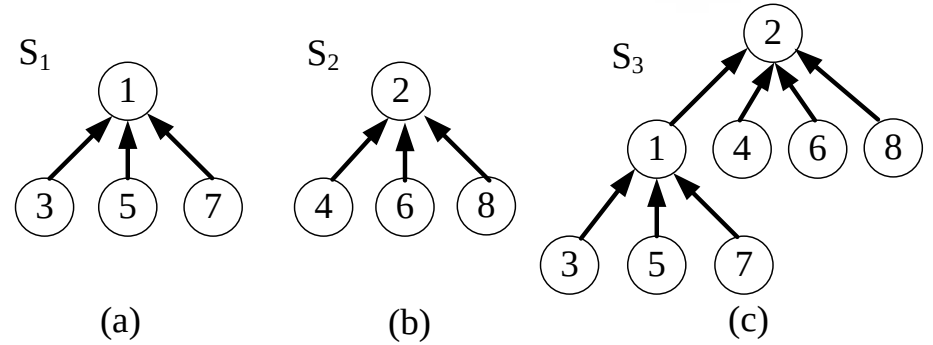
- There is a set S, having n elements, and a set R, having r relations, which are defined based on S. And x, y, z, are the elements of the set S.
- The relation R will be an equivalence relation, if and only if the following conditions are true:
  a) $(x, x) \in R$ for all (**reflexivity**)
  b) $(y, x) \in R$ if and only if $(x, y) \in R$ (**symmetry**)
  c) If $(x, y) \in R$ and $(y, z) \in R$, then $(x, z) \in R$ (**transitivity**)
- If $(x, y) \in R$, we say elements x and y are equivalent

Equivalence Classes

- Equivalence class is the largest set consisting of elements which are equivalent to each other. The 'largest' means that there is no other element equivalent to any element in the set.

- An equivalence class derived from $x \in S$ based on the relation R
  - $[x]_R = \{y \mid y \in S \wedge xRy\}$
  - R partition S into r disjoint sets $S_1, S_2, \ldots S_r$, and the union of these sets equals to S

Use tree to represent the Union/Find of equivalence classes



- Use a tree to represent a set
  - The set can be represented by the root node
  - If two nodes are in the same tree, they belong to the same set
- The representation of tree
  - Store in a static pointer array
  - A node only needs to store the information of its parent node

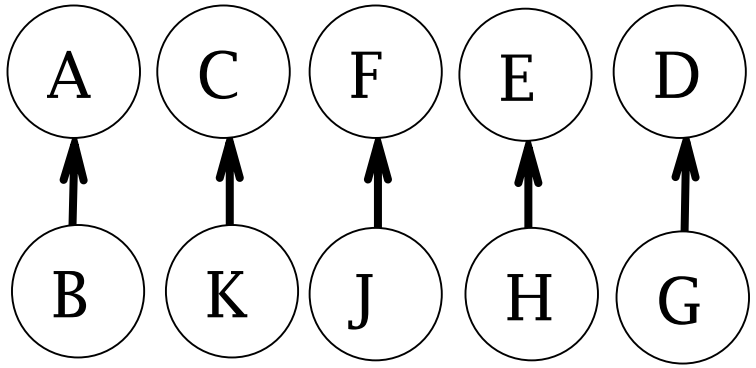
UNION/FIND Algorithm(1)

Process these 5 equivalence pairs (A, B),

(C, K) , (J, F), (H, E), (D, G)

	0		2				4	5	6
A	B	C	K	D	E	F	G	H	J
0	1	2	3	4	5	6	7	8	9

(A,B)(C,K)(J,F)(E,H)(D,G)

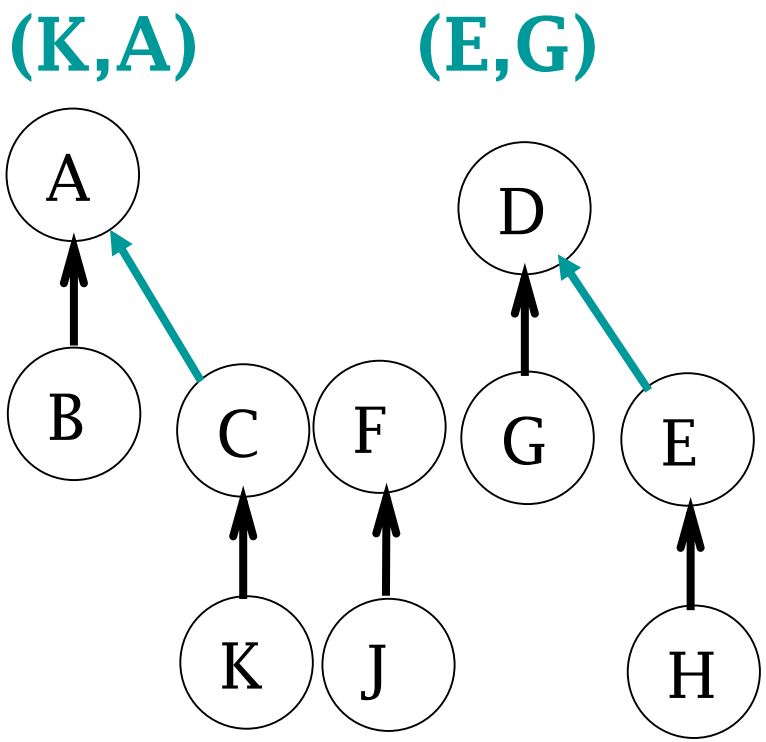


UNION/FIND Algorithm(1)

Then, process two equivalence pairs (K, A) and (E, G)

The root of the tree that K belongs to is C, A itself is a root node, and A ! =C, so merge these two trees.

	0	0	2		4		4	5	6
A	B	C	K	D	E	F	G	H	J
0	1	2	3	4	5	6	7	8	9



6.2 Linked Storage Structure of Tree

Parent pointer representation and Union/Find algorithm representation

```
template<class T>
class ParTreeNode  {                                  //Definition of tree node
private:
  Tvalue;                                             //The value of the node
  ParTreeNode<T>* parent;                             //The parent node pointer
  int nCount;                                         //The number of the nodes in the set
public:
  ParTreeNode();                                      //Constructor
  virtual ~ParTreeNode(){};                           //Destructor
  TgetValue();                                        //Return the value of the node
  void  setValue(const T& val);                       //Set the value of the node
  ParTreeNode<T>*  getParent();                       //Return the parent node pointer
  void  setParent(ParTreeNode<T>* par);               //Set the parent node pointer
  int  getCount();                                    //Return the number of nodes
  void  setCount(const int count);                    //Set the number of nodes
};
```

6.2 Linked Storage Structure of Tree

Parent pointer representation and Union/Find algorithm representation

```
template<class T>
class ParTree  {                                  // Definition of the tree
public:
  ParTreeNode<T>* array;                          // the array used to store the tree node
  int Size;                                       // the size of the array
  ParTreeNode<T>*
  Find(ParTreeNode<T>* node) const;               // Find the root node of "node"
  ParTree(const int size);                        // Constructor
  virtual ~ParTree();                             // Destructor
  void Union(int i,int j);                        // Union set i and j, and merge them
                                                  //    into the same subtree
  bool Different(int i,int j);   // Check if node i and j belong to the same tree
};
```

Parent pointer representation and Union/Find algorithm representation

```
template <class T>
ParTreeNode<T>*
ParTree<T>::Find(ParTreeNode<T>* node) const
{
  ParTreeNode<T>* pointer=node;
  while ( pointer->getParent() != NULL )
    pointer=pointer->getParent();
  return pointer;
}
```

Parent pointer representation and Union/Find algorithm representation

```
template<class T>
void ParTree<T>::Union(int i,int j) {
ParTreeNode<T>* pointeri = Find(&array[i]);      // find the root node of node i
ParTreeNode<T>* pointerj = Find(&array[j]);      // find the root node of node j
if (pointeri != pointerj) {
  if(pointeri->getCount() >= pointerj->getCount()) {
        pointerj->setParent(pointeri);
        pointeri->setCount(pointeri->getCount() +
                           pointerj->getCount());
  }
  else {
        pointeri->setParent(pointerj);
        pointerj->setCount(pointeri->getCount() +
                           pointerj->getCount());
   }
} }
```

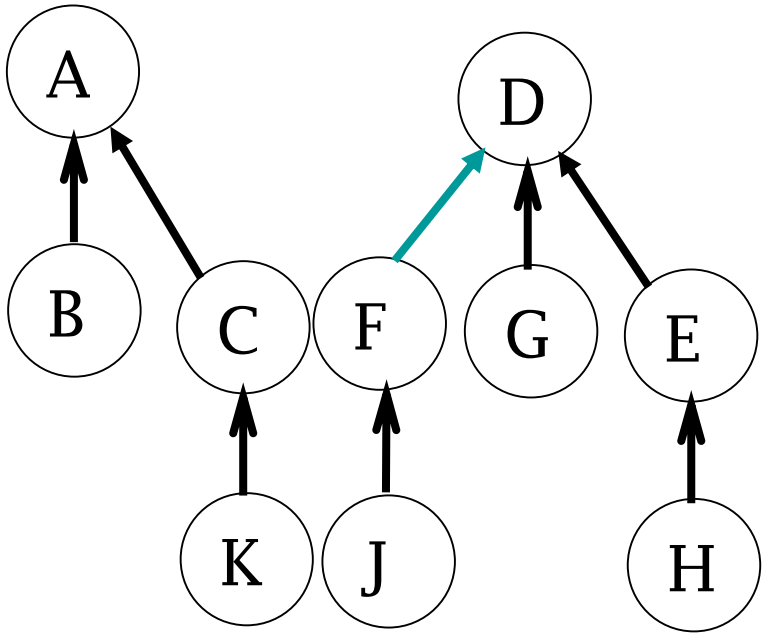
UNION/FIND Algorithm(2)

Use weighted union rule to process (H, J)

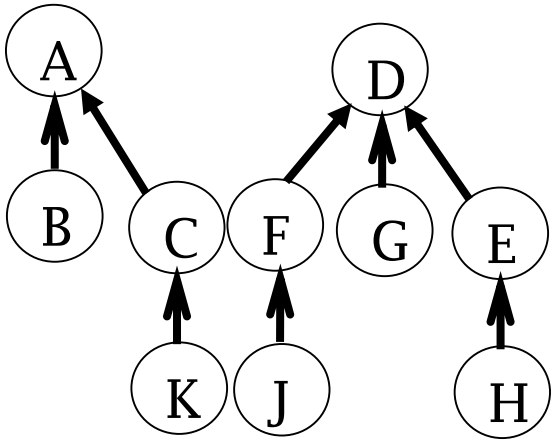
According to weighted union rule, the number of nodes of the tree whose root node is F is smaller, so let F point to D

	0	0	2		4	4	4	5	6
A	B	C	K	D	E	F	G	H	J
0	1	2	3	4	5	6	7	8	9

(H,J)



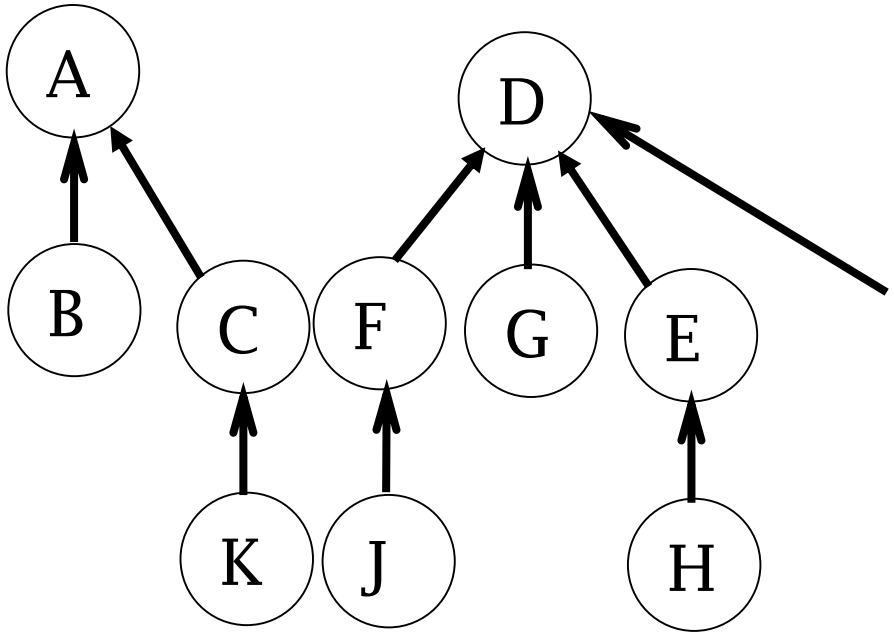
Path compression



- Find X
  - Assume that X finally reaches the root node R
  - Along the path from X to R, make parent pointer of every node point to R
- Low altitude trees come out

UNION/FIND Algorithm(3)

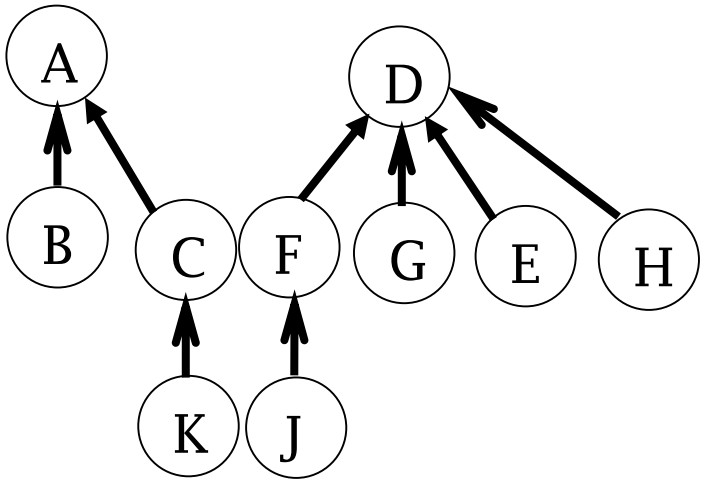
Use path compression to deal with Find(H)



	0	0	2		4	4	4	4	6
A	B	C	K	D	E	F	G	H	J
0	1	2	3	4	5	6	7	8	9

Path compression

```
template <class T>
ParTreeNode<T>*
ParTree<T>::FindPC(ParTreeNode<T>* node) const
{
  if (node->getParent() == NULL)
    return node;
  node->setParent(FindPC(node->getParent()));
  return node->getParent();
}
```



Path compression make the expenditure of Find close to a constant

- Weight + path compression
- The expenditure of n Find operations for n nodes is O(nα(n)), which is about Θ(nlog*n)
  - α(n) is the inverse of univariate Ackermann function, its growth rate is slower than logn, but not equals to constant
  - log*n is the number of operations that calculate log(n) before n = logn ≤ 1
  - log*65536 = 4 (4 log operations)
- The algorithm of find needs at most n Find operations, so it is very close to Θ(n)
  - In practical applications, α(n) is usually smaller than 4

Thinking

- Can we use dynamic pointer representation to accomplish parent pointer representation?
- Consult books or websites, read different optimizations of weighted union rule and path compression. Discuss their differences, advantages and disadvantages.

Data Structures and Algorithms

Thanks

the National Elaborate Course (Only available for IPs in China)
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

Ming Zhang, Tengjiao Wang and Haiyan Zhao

Higher Education Press, 2008.6 (awarded as the "Eleventh Five-Year" national planning textbook)