Lecture 6: Disjoint Set and Binary Search

Dr. Tsung-Wei Huang
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT

Union and Find (or Disjoint Set)

Disjoint Set

- We have a collection of disjoint sets of elements. Each set is identified by a representative element. We want to perform union operations and tell which element belongs to which set so we can know the connectivity among elements. Formally, we have the following operations.

Basic Operation

- MAKE-SET(x): Create new set {x} with one element x.
- UNION(x, y): x and y are elements of two sets. Union them into a single set. Choose a representative for the merged set.
- FIND-SET(x): return the representative of the set containing x.

Example

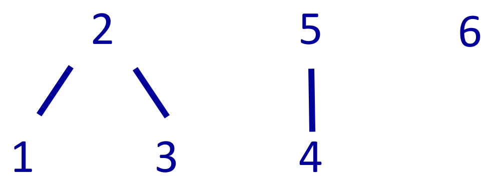
- 4 elements, 1, 2, 3, and 4

MAKE-SET(1)		{1}
MAKE-SET(2)		{2}
MAKE-SET(3)		{3}
MAKE-SET(4)		{4}
FIND(3)	**(returns 3)**	
FIND(2)	**(returns 2)**	
UNION(1,2)	**(representative 1, say)**	{1,2}
FIND(2)	**(returns 1)**	
FIND(1)	**(returns 1)**	
UNION(3,4)	**(representative 4, say)**	{3,4}
FIND(4)	**(returns 4)**	
FIND(3)	**(returns 4)**	
UNION(1,3)	**(representative 4, say)**	{1,2,3,4}
FIND(2)	**(returns 4)**	
FIND(1)	**(returns 4)**	
FIND(4)	**(returns 4)**	
FIND(3)	**(returns 4)**	

Disjoin Set Implementation

❑ Forest Implementation

Here we represent each set as a tree, and the **representative** is the **root** . For example, the following forest represents the set {1,2,3}, {4,5}, {6}：



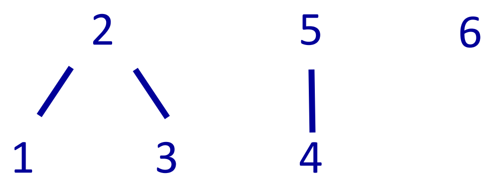
Implementation

MAKE-SET(x) Create a tree
FIND-SET(x) Return the root
UNION(x,y) Combine two trees

1	2	3	4	5	6
2	2	2	5	5	6

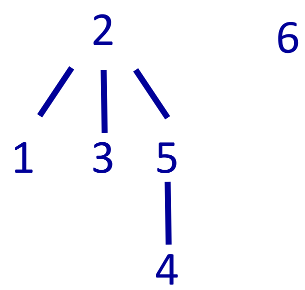
Disjoin Set Implementation

❑ Forest Implementation



1	2	3	4	5	6
2	2	2	5	5	6

Thus we would get the following form UNION(1,4)



1	2	3	4	5	6
2	2	2	5	2	6

This representation enables simple and efficient array operations by repointing the parent of a set to another set

Disjoin Set Implementation

- **Path compression and rank**
  - These are refinements of the forest representation which make it run significantly faster
    - FIND-SET: Do path compression
    - UNION: Use ranks

  - “Path compression” means that when we do FIND-SET(X), we make all nodes encountered point directly to the representative element for x. Initially, all elements have rank 0. The ranks of representative elements are updated so that if two sets with representatives of the same rank are merged, then the new representative is incremented by one.

Path Compression

MAKE-SET(1) … MAKE-SET(6)

1 2 3 4 5 6 RANKS = 0

UNION(1,2) UNION(4,5)

2 3 5 6 RANK(2)=1

RANK(5)=1

1 4

UNION(1,3)

2 5 6 RANK(2)=1

RANK(5)=1

1 3 4

UNION(5,6)

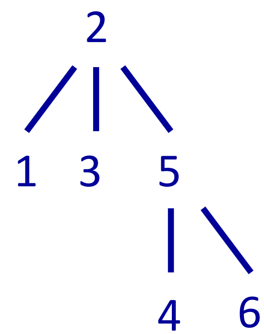
2 5 RANK(2)=1

RANK(5)=1

1 3 4 6

Path Compression

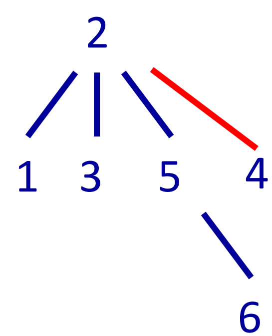
UNION(4,3)



RANK(2)=2

RANK(5)=1

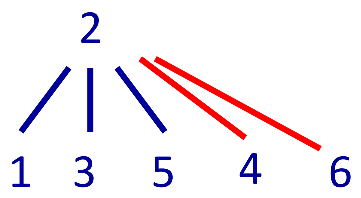
FIND(4) (path compression)



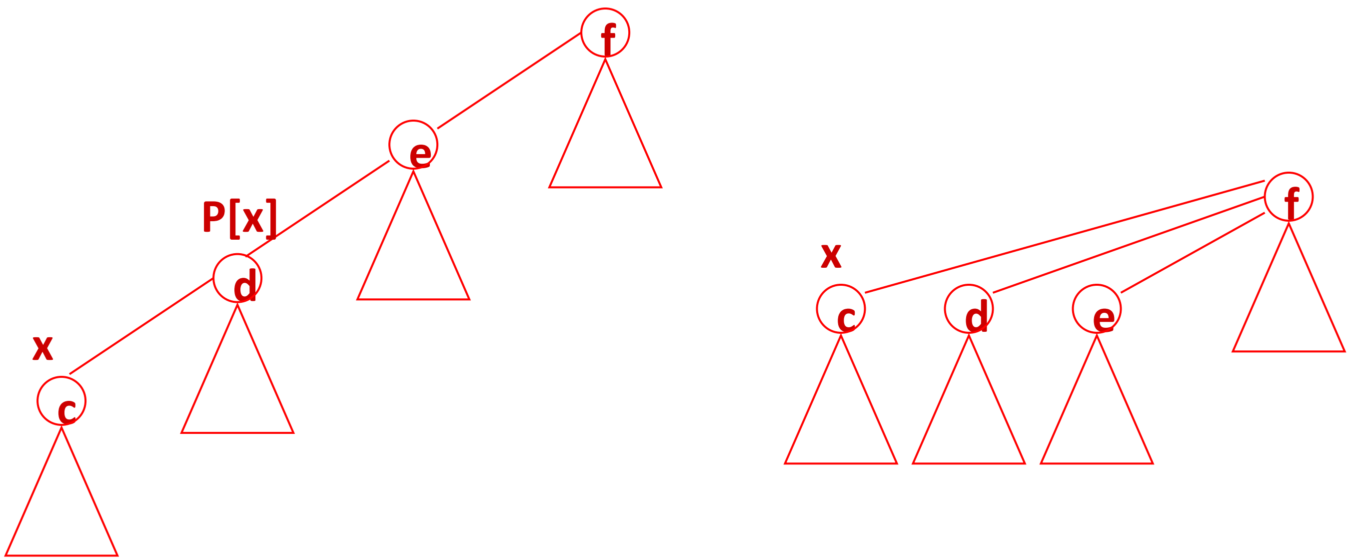
Path Compression

FIND(3) no change

FIND(6) (path compression)



Self-adjustment Data Structure



Code

- MakeSet and Union

```
void MakeSet(int x)
{
  p[x]    = x;
  rank[x] = 0;
}

void Union(int x,int y)
{
    Link(FindSet(x),FindSet(y));
}
```

Code

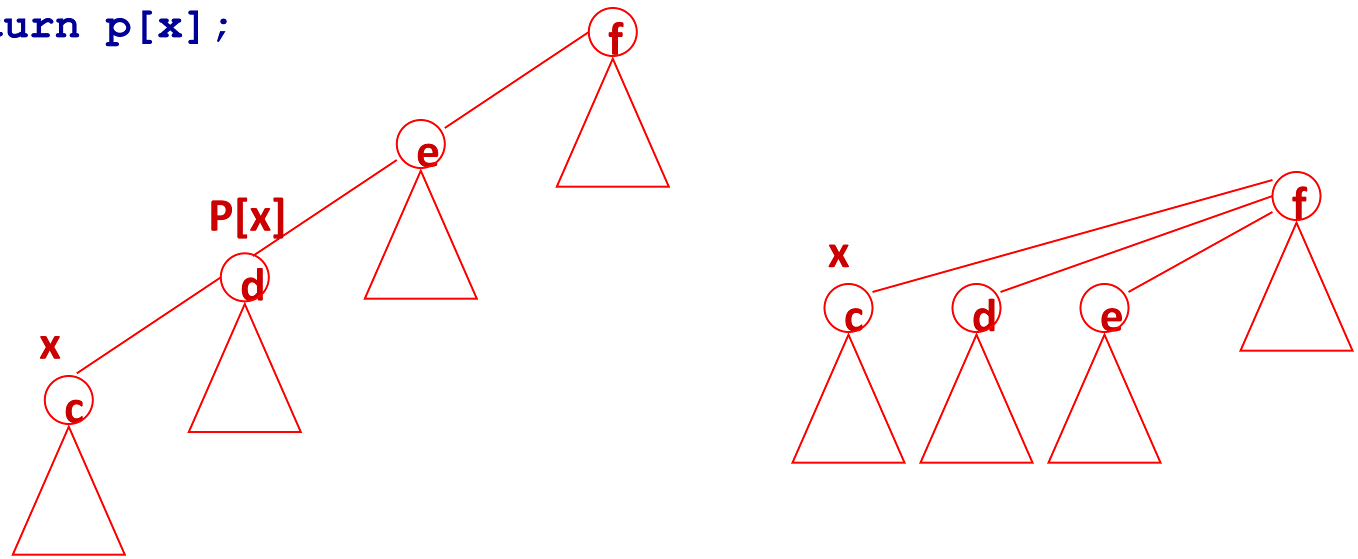
- Link

```
void Link(int x,int y)
{
    if(rank[x]>rank[y])
        p[y] = x;
    else
    {
        p[x] = y;
        if(rank[x]==rank[y])
            rank[y]++;
    }
}
```

Code

- FindSet

```
int FindSet(int x)
{
    if(x!=p[x])
        p[x] = FindSet(p[x]);
    return p[x];
}
```



Example – Computer Connectivity

Problem Description

Consider a set of N computers numbered from 1 to N, and a set S of M computer pairs, where each pair (i,j) in S indicates that computers i and j are connected. The connectivity rule says that if computers i and j are connected, and computers j and k are connected, then computers i and k are connected, too, no matter whether (i,k) or (k,i) is in S or not.

Based on S and the connectivity rule, the set of N computers can be divided into a number of groups such that for any two computers, they are in the same group if and only if they are connected. Note that if a computer is not connected to any other one, itself forms a group.

A group is said to be largest if the number of computers in it is maximum among all groups. The problem asks to count how many computers there are in a largest group.

Example – Computer Connectivity

I/O Description

The first line of the input file contains the number of test cases. For each test case, the first line consists of N and M, where N is the number of computers and M is the number of computer pairs in S. Each of the following M lines consists of two integers i and j ($1 \le i \le N$, $1 \le j \le N$, $i = j$) indicating that (i,j) is in S. Note that there could be repetitions among the pairs in S.

Example – Computer Connectivity

Sample Input

1

3 4

1 2

3 2

2 3

1 2

Sample Input

3

Binary Search

- Binary Search
  - Search the index of a value in a sorted array
- Basic Operation
  - Work normally on the index of the item
  - Find MAX or Find MIN
  - Set the function valid to return true on your criteria
  - Set the while loop (beg, end, mid)
- Index types
  - Integer (most cases)
  - double

Binary Tree Example

- Search the value 33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑														↑
lo														hi

Binary Tree Example

- Search the value 33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑							↑							↑
lo							mid							hi

Binary Tree Example

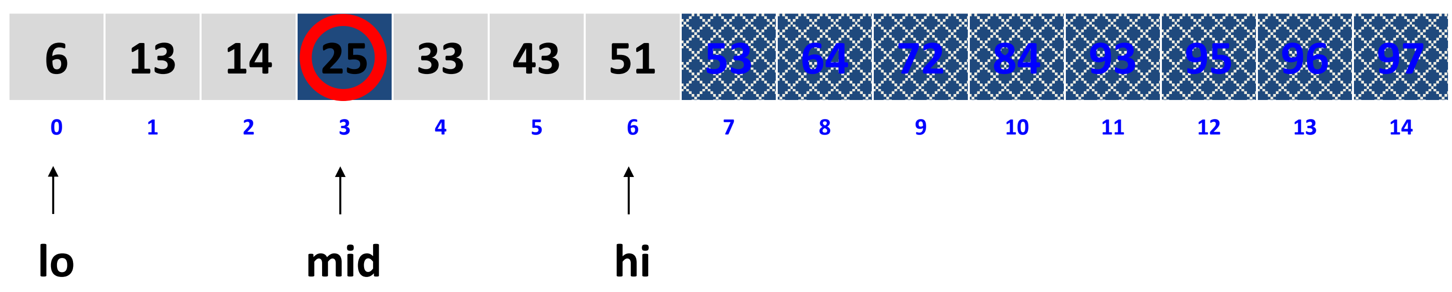
- Search the value 33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

lo (index 0) hi (index 6)

Binary Tree Example

- Search the value 33



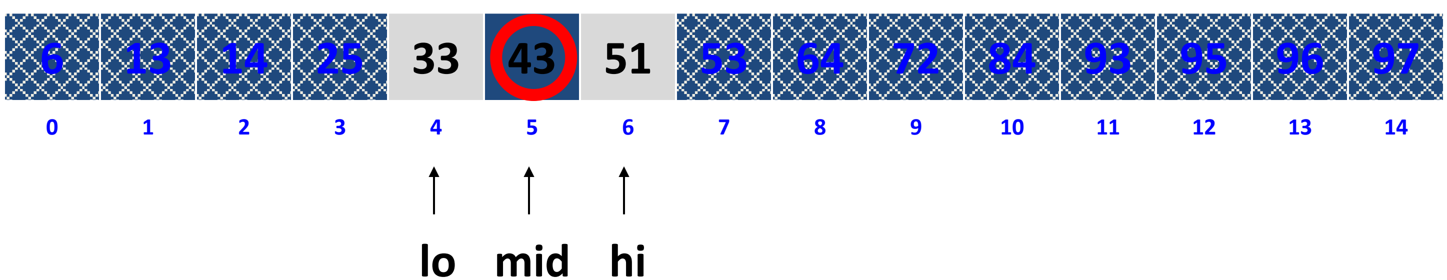
Binary Tree Example

- **Search the value 33**

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				lo		hi								

Binary Tree Example

- Search the value 33



Binary Tree Example

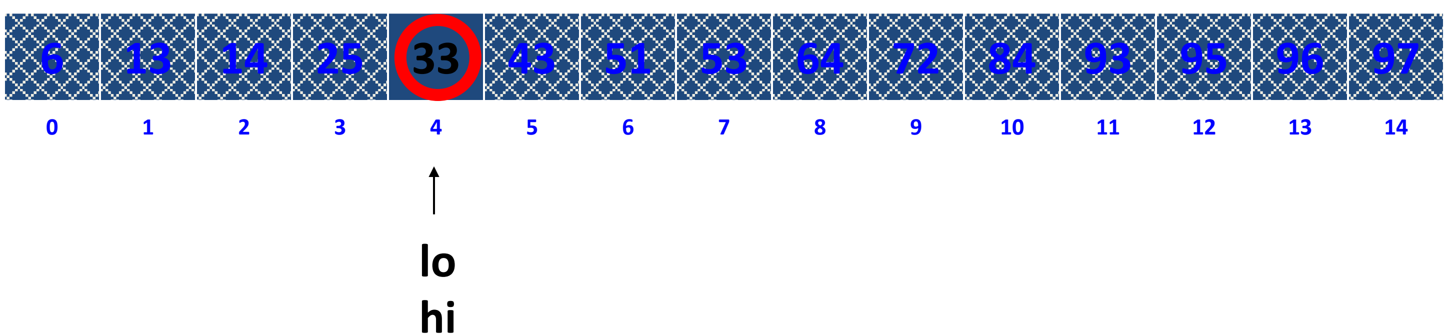
- Search the value 33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

lo
hi

Binary Tree Example

- Search the value 33



Golden Formula – Find min

```
void FindMin()
{
	int beg, end, mid, best = INT_MAX;
	do {
		mid = (beg+end)/2;
		if(valid(mid))
		{
			best = min(best, mid);
			end = mid;
		}
		else
			beg = mid + 1;
	}while(beg < end);

	/* now the answer is in the best, if the value is INT_MAX means
	 * no feasible solution. Otherwise it is the minimum value. */
}
```

Include this to your software toolbox!

Golden Formula – Find Max

```
void FindMax()
{
	int beg, end, mid, best = 0;

	do {
		mid = (beg+end+1) / 2;
		if(valid(mid)) {
			best = min(best, mid);
			beg = mid;
		}
		else end = mid - 1;

	}while(beg < end);

	/* now the answer if the best, if the value  is the 0(user-defined minimum
	   value), means no solution. Otherwise, it is the maximum value. */
}
```

Include this to your software toolbox!

Binary Search Example

- **A sorted array: 1, 2, 4, 7, 9, 11, 15, 17, 19, 31, 40**
- **Find the minimum value that > 13**
  - What about finding the minimum value that < 13
- **Find the maximum value that < 13**
  - What about finding the maximum value that > 13

minimum value that > 13

- mid = (0+10) / 2

0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	9	11	15	17	19	31	40
beg					mid					end

minimum value that > 13

- mid = (6+10) / 2

0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	9	11	15	17	19	31	40
						beg		mid		end

minimum value that > 13

- mid = (6+8) / 2

0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	9	11	15	17	19	31	40
						beg	mid	end		

minimum value that > 13

- mid = (6+7) / 2

0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	9	11	15	17	19	31	40
						beg mid	end			

minimum value that > 13

- mid = (6+6) / 2

0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	9	11	15	17	19	31	40
						beg end mid				

maximum value that < 13

- **mid = (0+10+1) / 2**

0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	9	11	15	17	19	31	40
beg					mid					end

maximum value that < 13

- mid = (5+10+1) / 2

0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	9	11	15	17	19	31	40
					beg			mid		end

maximum value that < 13

- **mid = (5+7+1) / 2**

0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	9	11	15	17	19	31	40
					beg	mid	end			

maximum value that < 13

- mid = (5+5+1) / 2

0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	9	11	15	17	19	31	40
					beg end mid					

Maximum i such that A[i] == 13

- mid = (0+10+1) / 2

0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	13	13	13	13	13	31	40
beg					mid					end

Maximum i such that A[i] == 13

- mid = (5+10+1) / 2

0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	13	13	13	13	13	31	40
					beg			mid		end

Maximum i such that A[i] == 13

- mid = (8+10+1) / 2

0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	13	13	13	13	13	31	40
								beg	mid	end

Maximum i such that A[i] == 13

- mid = (8+8+1) / 2

0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	13	13	13	13	13	31	40
								beg end mid		

What if A does not have 13?

- **mid = (0+10+1) / 2**
- **Cannot use equal comparison in the valid function**
  - Must transform $\leq$

0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	9	10	11	12	16	31	40
beg					mid					end

What if A does not have 13?

- **mid = (5+10+1) / 2**

0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	9	10	11	12	16	31	40
					beg			mid		end

What if A does not have 13?

- mid = (5+7+1) / 2

0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	9	10	11	12	16	31	40
					beg	mid	end			

What if A does not have 13?

- mid = (6+7+1) / 2

0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	9	10	11	12	16	31	40
						beg	end mid			

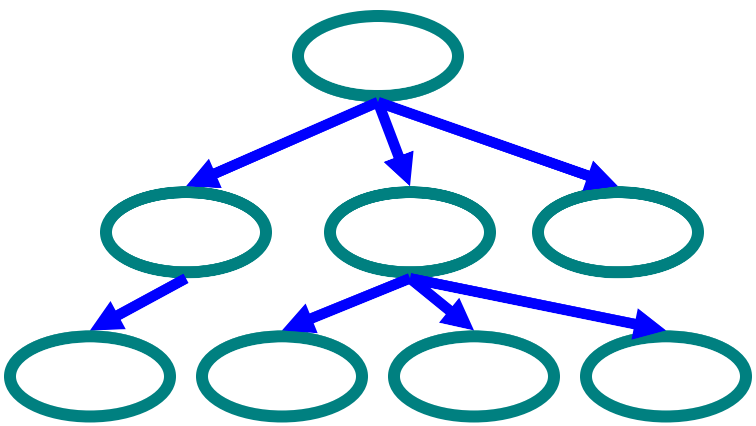
What if A does not have 13?

- mid = (7+7+1) / 2

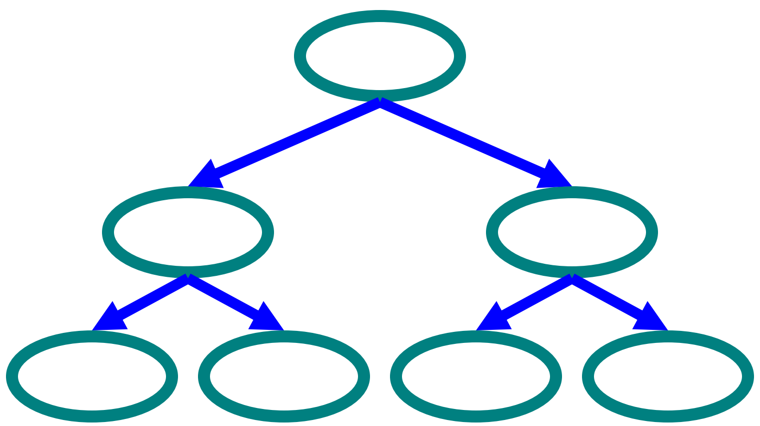
0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	9	10	11	12	16	31	40
							beg end mid			

Tree

- Tree
  - A data structure with N vertices and N-1 edge
  - A basic connected component of N vertices
  - An acyclic Graph
  - root, leaf, and inter node



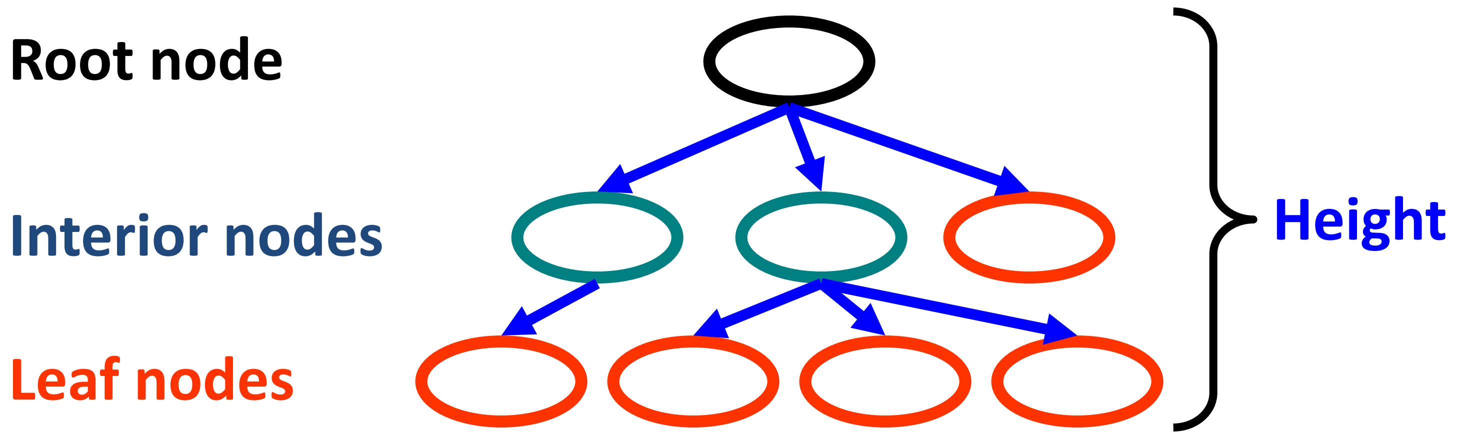
Tree



Binary Tree (deg=2)

Tree

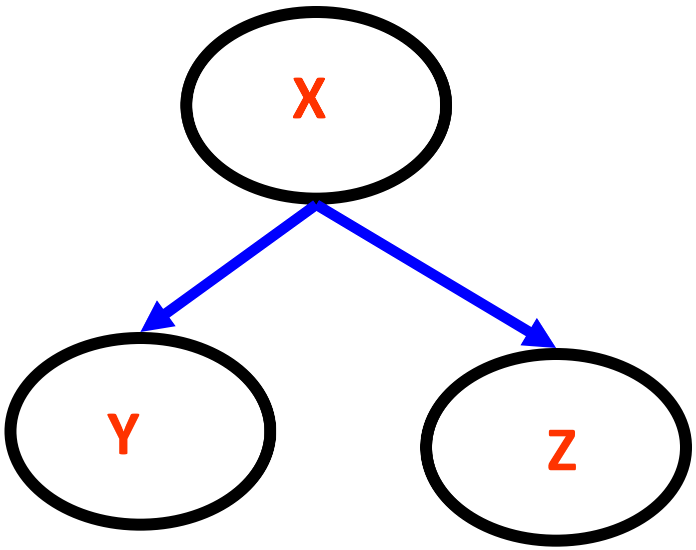
- **Terminology**
  - Root $\Rightarrow$ no parent
  - Leaf $\Rightarrow$ no child
  - Interior $\Rightarrow$ non-leaf
  - Height $\Rightarrow$ distance from root to leaf



Binary Search Tree

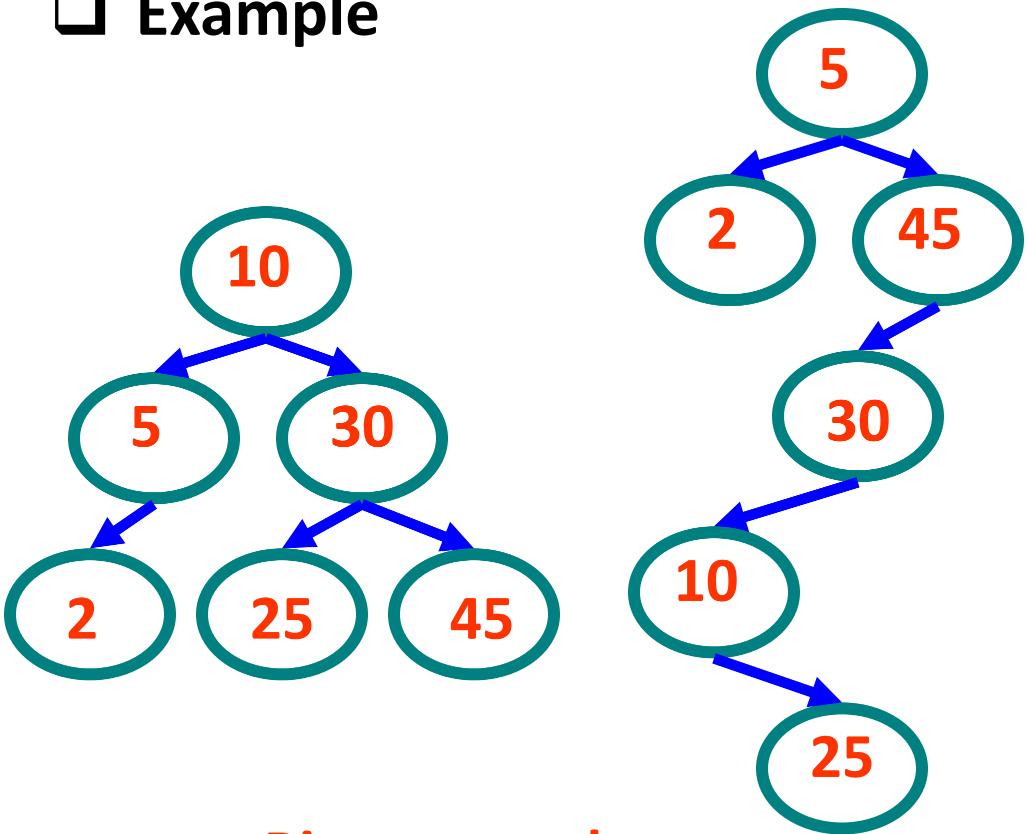
- **Key property**
  - Value at node
    - Smaller values in left subtree
    - Larger values in right subtree
  - Example
    - X > Y
    - X < Z

```
struct Node {
   int value;
   Node* right_child;
   Node* left_child;
}
```

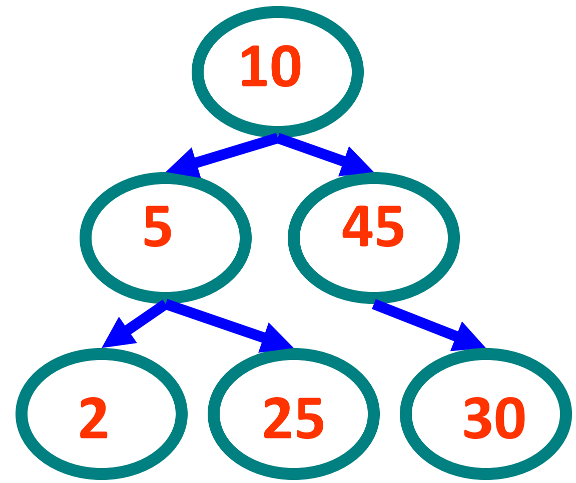


Binary Search Tree

- ❑ **Example**



Binary search trees



Not a binary search tree

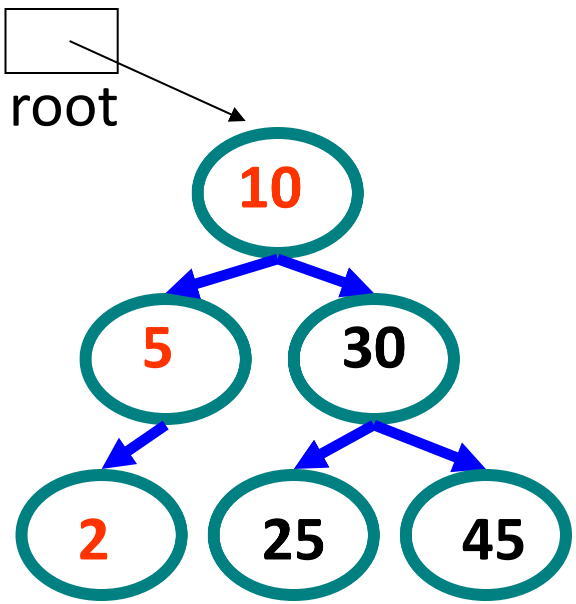
Binary Search Tree

- Implementation Example – Find an Element in the Tree

```
Node *Find( Node *n, int key) {
        if (n == NULL)                          // Not found
              return( n );
        else if (n->data == key)  // Found it
              return( n );
        else if (n->data > key)    // In left subtree
              return Find( n->left, key );
        else                                          // In right subtree
              return Find( n->right, key );
}
Node * n = Find( root, 5);
```

Binary Search Tree

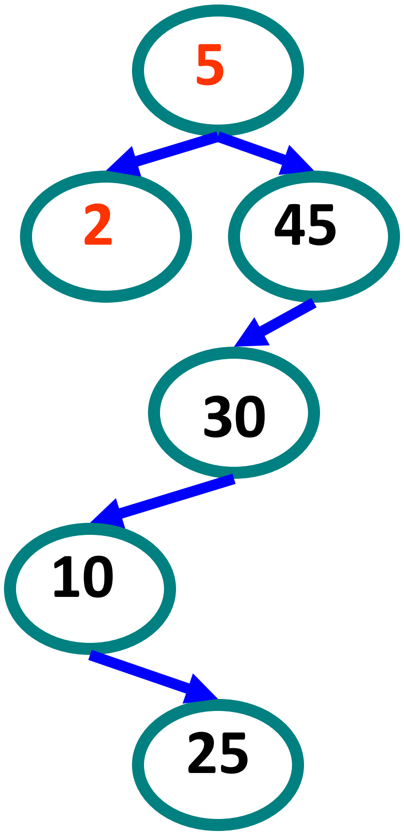
- Find (root, 2)



10 > 2, left

5 > 2, left

2 = 2, found

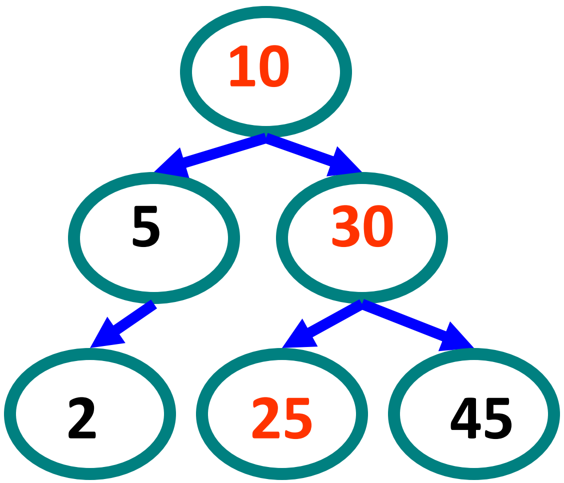


5 > 2, left

2 = 2, found

Binary Search Tree

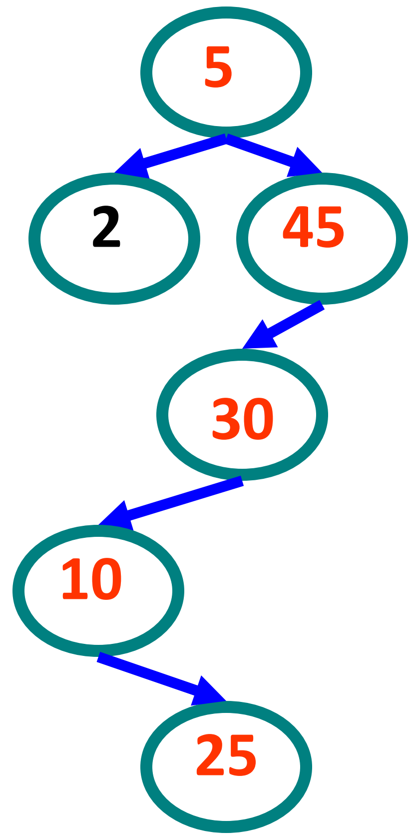
- Find (root, 25)



10 < 25, right

30 > 25, left

25 = 25, found



5 < 25, right

45 > 25, left

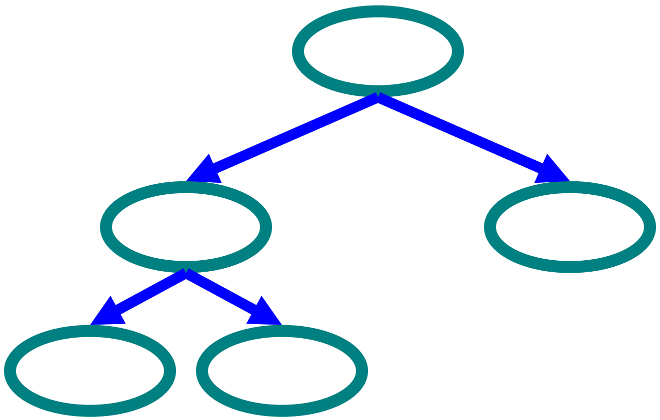
30 > 25, left

10 < 25, right

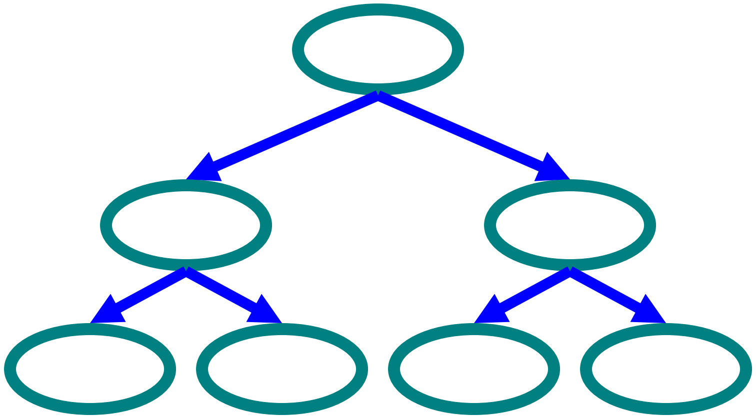
25 = 25, found

Complete Binary Tree

- Complete Binary Tree
  - Grow by Each Level

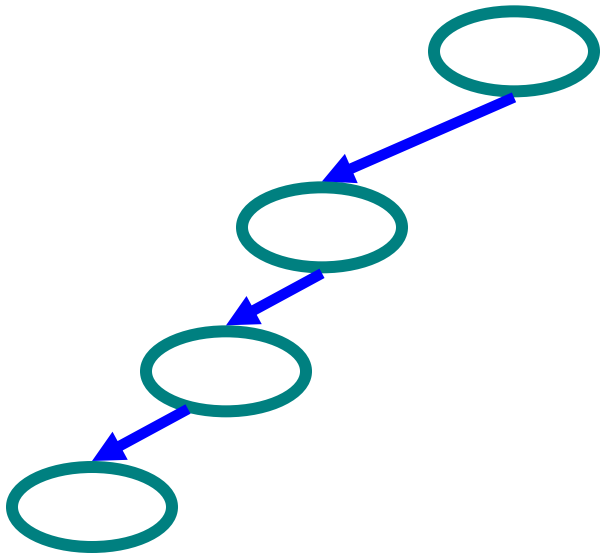


- Full Binary Tree
  - Leaf level is full



Skewed Binary Tree

- Skewed Tree



Time Complexity

- Time of search
  - Proportional to height of tree
  - Balanced binary tree
    - O(log(n)) time
  - Degenerated tree (skewed)
    - O(n) time
    - Like searching linked list / unsorted array

By How?

- **Use std::set and std::map**
  - Implemented balanced binary tree using “red-black tree” algorithms
- **Balanced binary tree is extremely difficult to implement right**
  - Rotation
  - Adjustment
  - …