Data Structures
資料結構

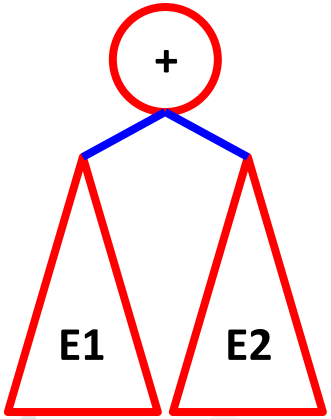
Trees – Part II

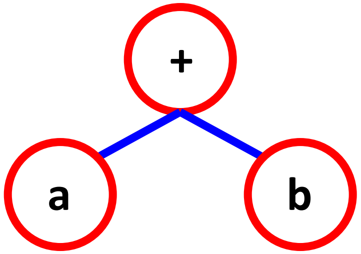
Department of Computer Science
National Tsing Hua University

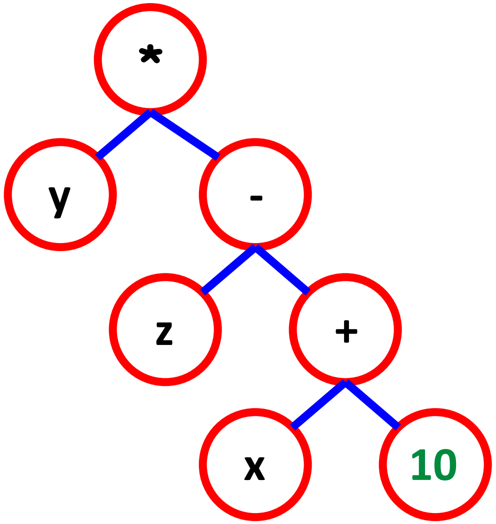
BINARY TREE APPLICATIONS

Expression Tree

- Given a regular expression, put **operands** at **leaf** nodes and **operators** at **nonterminal** nodes







Inorder	E1 + E2	a + b	y * (z – (x + 10))	**Infix notation**
Preorder	+ E1 E2	+ a b	* y – z + z 10	**Prefix notation**
Postorder	E1 E2 +	a b +	y z x 10 + - *	**Postfix notation**

Priority Queue

- In a **priority queue**, the element to be processed/deleted is the one with **highest** (or **lowest**) priority
- Operations
  - Get the max/min element
  - Insert an element to the priority queue
  - Delete element with max/min priority

ADT : Priority Queue

```
template < class T >
class MaxPQ
{
public:
    MaxPQ();
   ~MaxPQ();

    // Check if PQ is empty
    bool IsEmpty() const;
    // Return reference to the max element
    T& Top() const;
    // Add an element to the PQ
    void Push(const T&);
    // Delete element with max priority
    void Pop();
private:
    // Data representation here
    // …
};
```

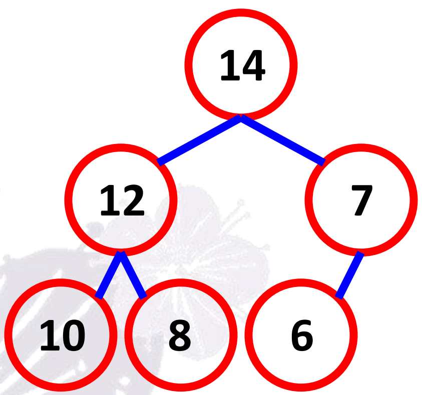
PQ Representations

- Unsorted linear list
  - Array, chain,..,etc
- Sorted linear list
  - Sorted array, sorted chain,...,etc
- Heap

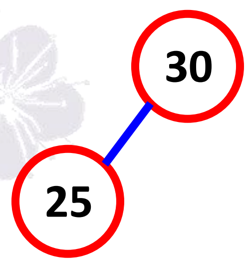
	Top()	Push()	Pop()
Unsorted linear list	O(n)	O(1)	O(n)
Sorted linear list	O(1)	O(n)	O(1)
Heap	O(1)	O(logn)	O(logn)

Max Heap

- Definition: A ***max (min) tree*** is a tree in which the key value in each node is ***no smaller*** (***larger***) than the key values in its children (if any). A ***max (min) heap*** is a ***complete binary tree*** that is also a ***max (min) tree***.



Max Heap

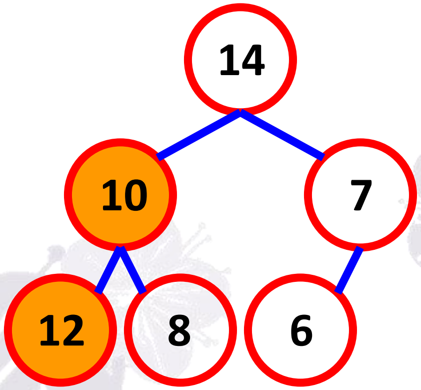


Max Heap

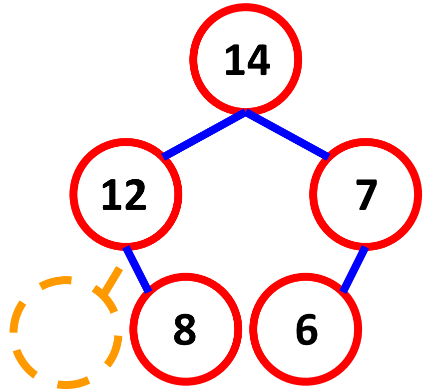
Max/Min Heap

Max Heap

- Definition: A ***max (min) tree*** is a tree in which the key value in each node is ***no smaller*** (***larger***) than the key values in its children (if any). A ***max (min) heap*** is a ***complete binary tree*** that is also a ***max (min) tree***.



Not a heap
(12 > 10)



Not a heap
(Not a complete binary tree)

Max Heap : Representation

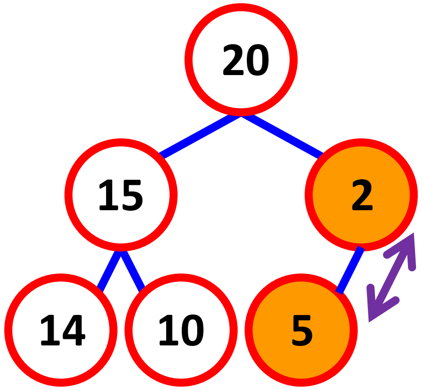
- Since the heap is a complete binary tree, we could adopt "**Array Representation**" as we mentioned before!
- Let node i be in position i (array[0] is empty)
  - **Parent(i) = $\lfloor i / 2 \rfloor$** if i ≠ 1. If i=1, i is the root and has no parent.
  - **leftChild(i) = 2i** if 2i ≤ n. If 2i > n, the i has no left child.
  - **rightChild(i) = 2i+1** if 2i+1 ≤ n, if 2i+1 > n, the i has no right child.

ADT : Priority Queue

```
template < class T >
class MaxPQ
{
public:
     MaxPQ();
    ~MaxPQ();

     // Check if PQ is empty
     bool IsEmpty() const;
     // Return reference to the max element
     T& Top() const;
     // Add an element to the PQ
     void Push(const T&);
     // Delete element with max priority
     void Pop();
private:
     T* heap          // Element array
     int heapSize; // # of elements
     int capacity; // size of the array "heap"
};
```

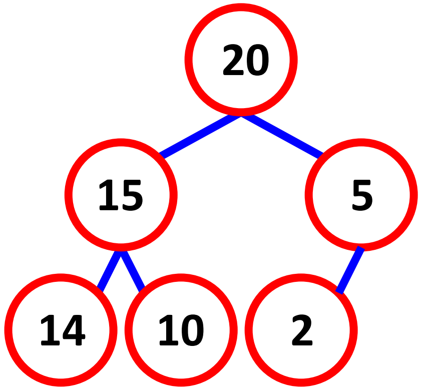
Max Heap : Insert

- Insert (5)
- Make sure it is a complete binary tree
- Check if the new node is greater than its parent
- If so, swap two nodes



Max Heap : Insert

- Insert (5)
- Make sure it is a complete binary tree
- Check if the new node is greater than its parent
- If so, swap two nodes



Max Heap : Insert Codes

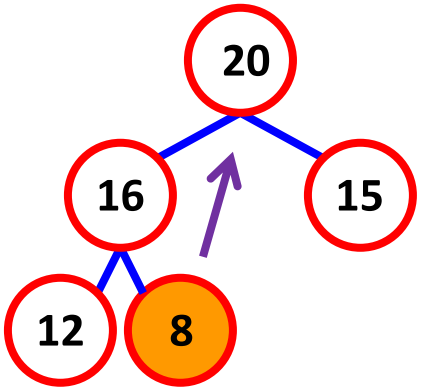
```
template < class T >
void MaxPQ<T>::Push(const T& e)
{ // Insert e into max heap
  // Make sure the array has enough space here…
  // …
  int currentNode = ++heapSize;
  while(currentNode != 1 && heap[currentNode/2] < e)
  { // Swap with parent node
    heap[currentNode]=heap[currentNode/2];
    currentNode /= 2; // currentNode now points to parent
  }
  heap[currentNode]=e;
}
```

Time Complexity

Travel at most the height of a tree, therefore is O(logn)

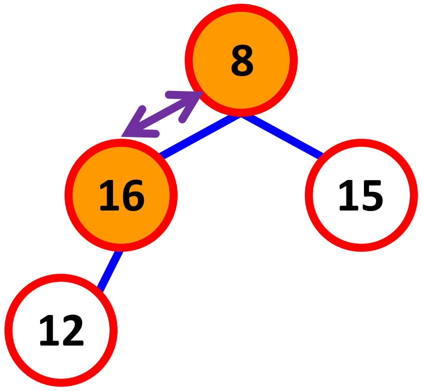
Max Heap : Delete

- Always delete the root
- Move the last element to the root (maintain a complete binary tree)



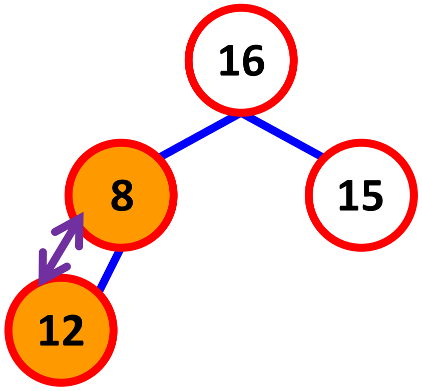
Max Heap : Delete

- Always delete the root
- Move the last element to the root (maintain a complete binary tree)
- Swap with larger and largest child (if any)



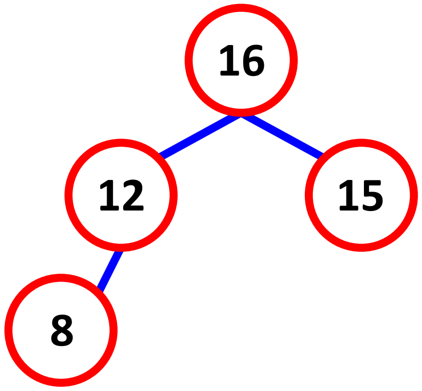
Max Heap : Delete

- Always delete the root
- Move the last element to the root (maintain a complete binary tree)
- Swap with larger and largest child (if any)
- Continue step 3 until the max heap is maintained (trickle down)



Max Heap : Delete

- Always delete the root
- Move the last element to the root (maintain a complete binary tree)
- Swap with larger and largest child (if any)
- Continue step 3 until the max heap is maintained (trickle down)



Max Heap : Delete Codes

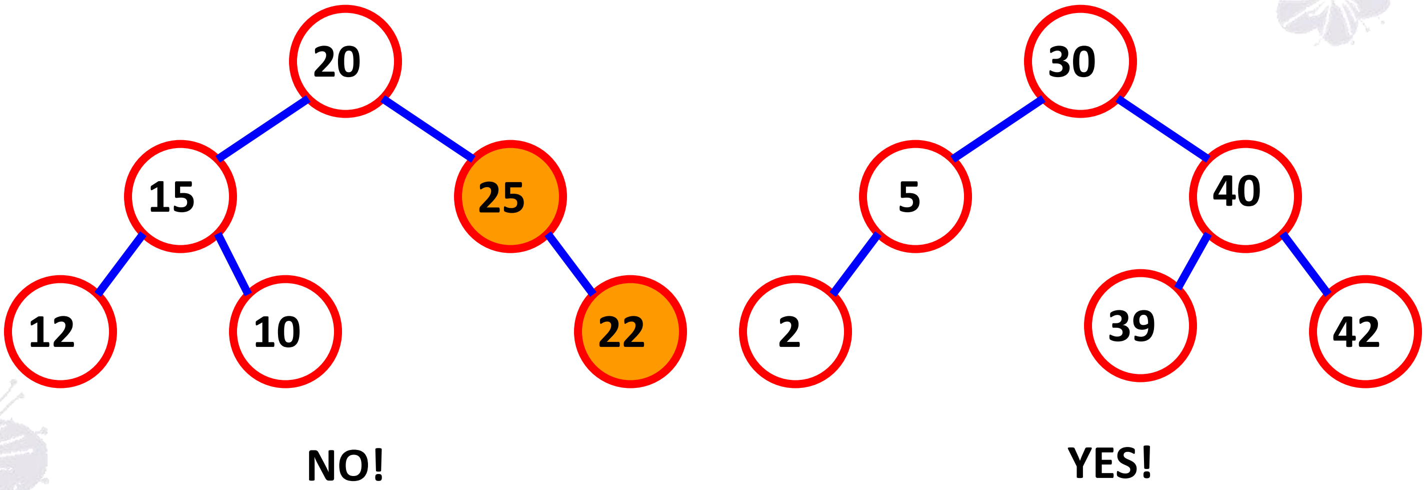
```
template < class T >
void MaxPQ<T>::Pop()
{ //Delete max element
  if(IsEmpty()) throw "Heap is empty";
  heap[1].~T(); // delete max element (always the root!)
  // Remove the last element from heap
  T lastE = heap[heapSize--];

  // trickle down
  int currentNode = 1; // root
  int child = 2; // A child of currentNode
  while(child <= heapSize) {
    // Set child to larger child of currentNode
    if (child < heapSize && heap[child] < heap[child + 1]) child++;

    // Can we put lastE in currentNode?
    if (lastE >= heap[child]) break; // Yes!

    // No!
    heap[currentNode] = heap[child]; // Move child up
    currentNode = child; child *=2;  // Move down a level
  }
  heap[currentNode] = lastE;
}
```

Time Complexity = Height of tree = O(logn)

Binary Search Tree

- Definition: A ***binary search tree (BST)*** is a binary tree which satisfies the following properties:
  - Every element has a ***key*** and no two elements have the same key.
  - The keys (if any) in the **left subtree** are **smaller** than the key in the root
  - The keys (if any) in the **right subtree** are **larger** than the key in the root
  - The left and right subtrees are also BST

BST: Examples



Inorder traversal?

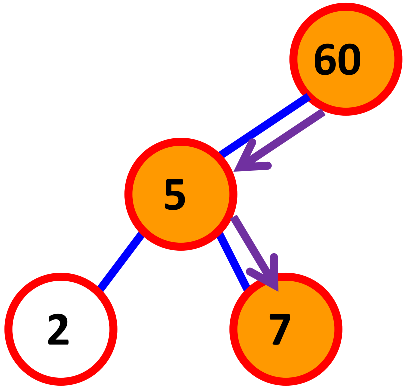
Inorder traversal of a BST will result in a sorted list

BST : Operations

- Search an element in a BST
- Search for the r^{th} smallest element in a BST
- Insert an element into a BST
- Delete max/min from a BST
- Delete an arbitrary element from a BST

BST : Search an Element

1. Search for key 7
2. Start from root
3. Compare the key with root
   – ‘<’ search the left subtree
   – ‘>’ search the right subtree
4. Repeat step 3 until the key is found or a leaf is visited



BST : Recursive Search Codes

```
template < class K, class E >
pair<K,E>* BST<K,E>::Get(const K& k)
{ // Search the BST for a pair with key k
  // If the this pair is found, return a pointer to this
  // pair, otherwise return 0
  return Get(root, k);
}

template < class K, class E >
pair<K,E>* BST<K,E>::Get(TreeNode<pair<K,E>>* p, const K& k)
{
  if(!p) return 0;
  if(k < p->data.first) return Get(p->leftChild, k);
  if(k > p->data.first) return Get(p->rightChild, k);
  return &p->data;
}
```

p->data.first = key

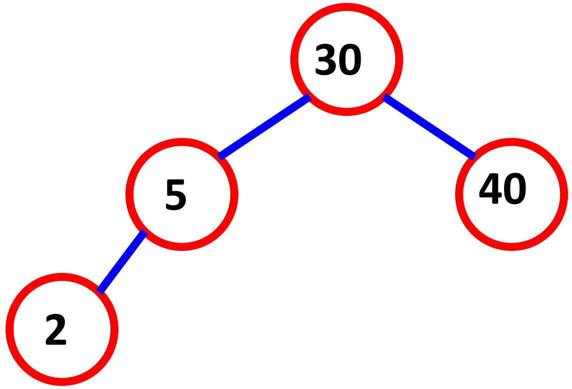
p->data.second = element

BST : Iterative Search Codes

```
template < class K, class E >
pair<K,E>* BST<K,E>::Get(const K& k)
{
  TreeNode < pair<K, E> > *currentNode = root;
  while (currentNode) {
     if (k < currentNode->data.first)
        currentNode = currentNode->leftChild;
     else if (k > currentNode->data.first)
        currentNode = currentNode->rightChild;
     else return & currentNode->data;
  }
  return NULL; // no match found
}
```

BST : Search an Element by Rank

- Definition of **rank**:
  - A ***rank*** of a node is its position in inorder traversal



Need r visits of nodes
Any faster ways?

Inorder traversal : 2 -> 5 -> 30 -> 40

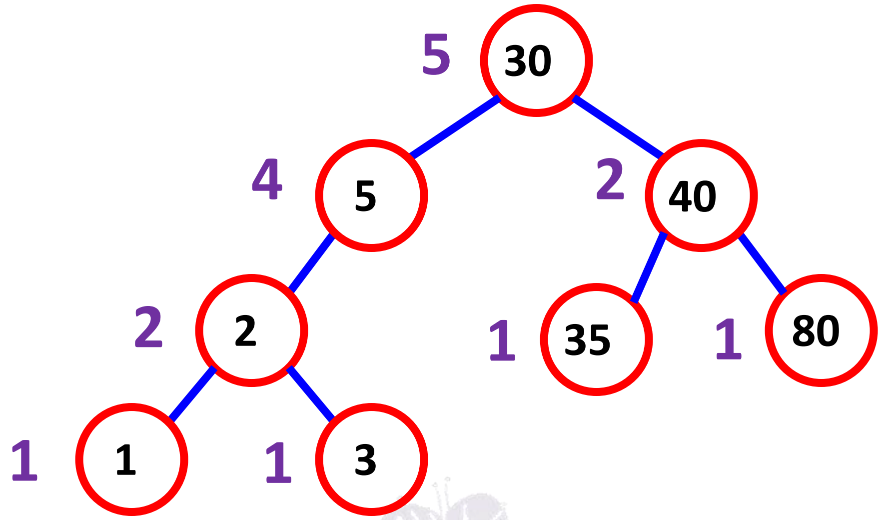
Rank : 1 2 3 4

Therefore, the r^{th} smallest element is the node with rank r

BST: Search by Rank - leftSize

To facilitate searching for rank-r element, we store the additional information, leftSize

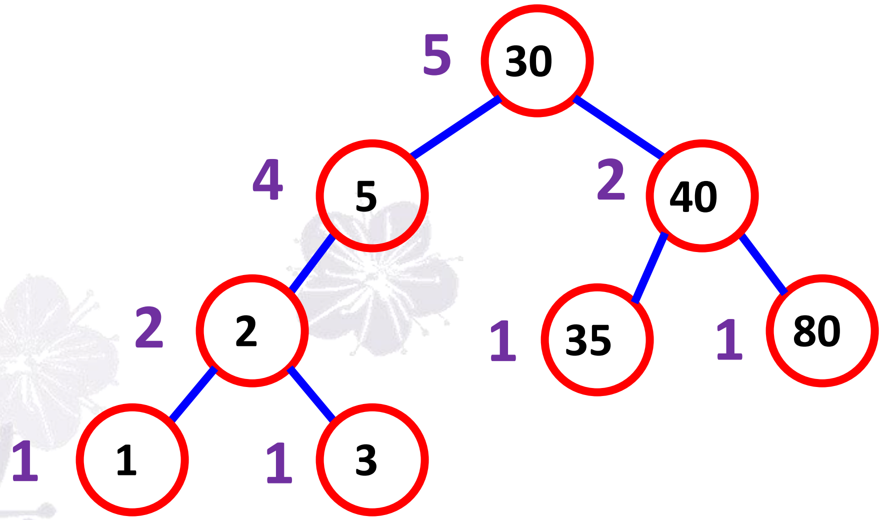
leftSize = 1 + # of nodes in left subtree



BST: Search by Rank - leftSize

If we are searching for the rank-r element, we perform:

1) Set currentNode = root
2) Consider 3 cases
   - leftSize > r: currentNode = left child; repeat 2)
   - leftSize < r: r = r – leftSize; currentNode = right child, repeat 2)
   - leftSize = r: bingo; break

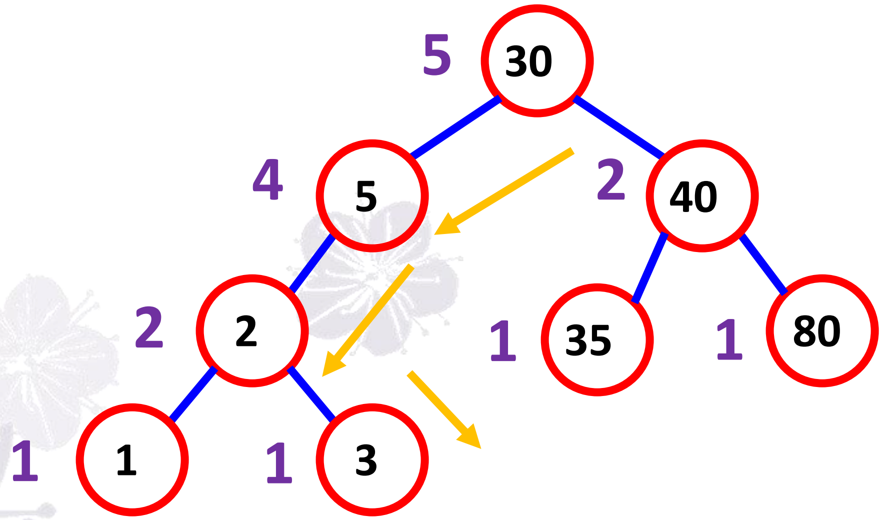


BST: Search by Rank - leftSize

If we are searching for the rank-r element, we perform:

1) Set currentNode = root
2) Consider 3 cases
   - leftSize > r: currentNode = left child; repeat 2)
   - leftSize < r: r = r – leftSize; currentNode = right child, repeat 2)
   - leftSize = r: bingo; break

Example: r=3

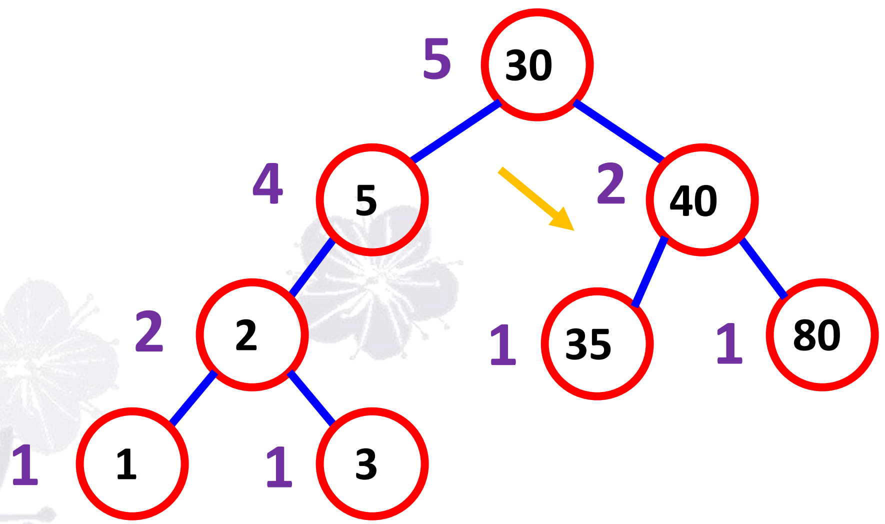


BST: Search by Rank - leftSize

If we are searching for the rank-r element, we perform:

1) Set currentNode = root
2) Consider 3 cases
   - leftSize < r: currentNode = left child; repeat 2)
   - leftSize > r: r = r – leftSize; currentNode = right child, repeat 2)
   - leftSize = r: bingo; break

Example: r=7

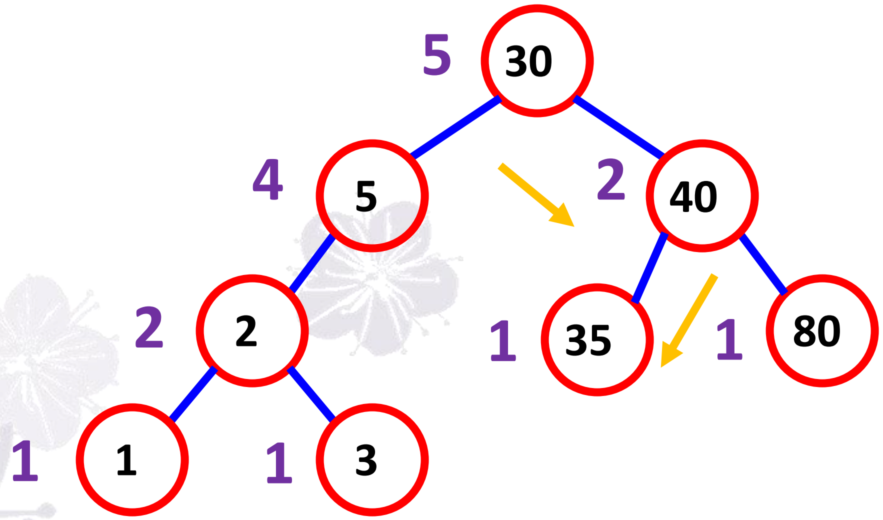


BST: Search by Rank - leftSize

If we are searching for the rank-r element, we perform:

1) Set currentNode = root
2) Consider 3 cases
   - leftSize > r: currentNode = left child; repeat 2)
   - leftSize < r: r = r – leftSize; currentNode = right child, repeat 2)
   - leftSize = r: bingo; break

Example: r=6



BST : Search by Rank Codes

- For each node, we store an additional information “leftSize” which is 1 + (# of nodes in the left subtree)

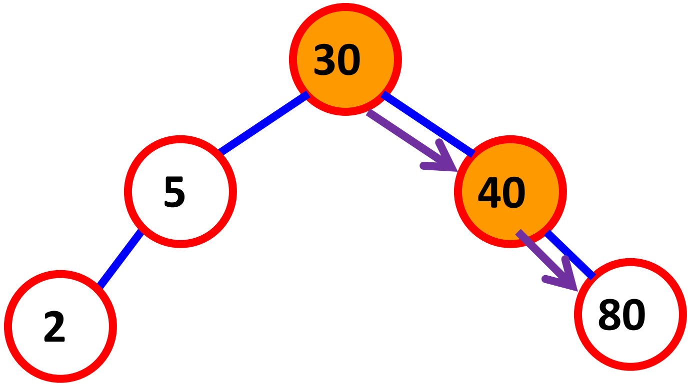
```
template < class K, class E >
pair<K,E>* BST<K,E>::RankGet(int r)
{ // Search BST for the rth smallest pair
  TreeNode<pair<K,E>>* currentNode = root;
  while(currentNode){
    if(r < currentNode->leftSize)
      currentNode = currentNode->leftChild;
    else if(r > currentNode->leftSize) {
      r -= currentNode->leftSize;
      currentNode = currentNode->rigthChild;
    }
    else return &currentNode->data;
  }
  return 0;
}
```

Question

- The r^{th} smallest element is the node with rank r
- What if we want to retrieve the r^{th} largest element?
- We can add a variable rightSize
- Or, we can simply perform a transformation …

BST : Insert

1. To insert an element with key 80
2. First we search for the existence of the element
3. If the search is unsuccessful, then the element is inserted at the point the search terminates

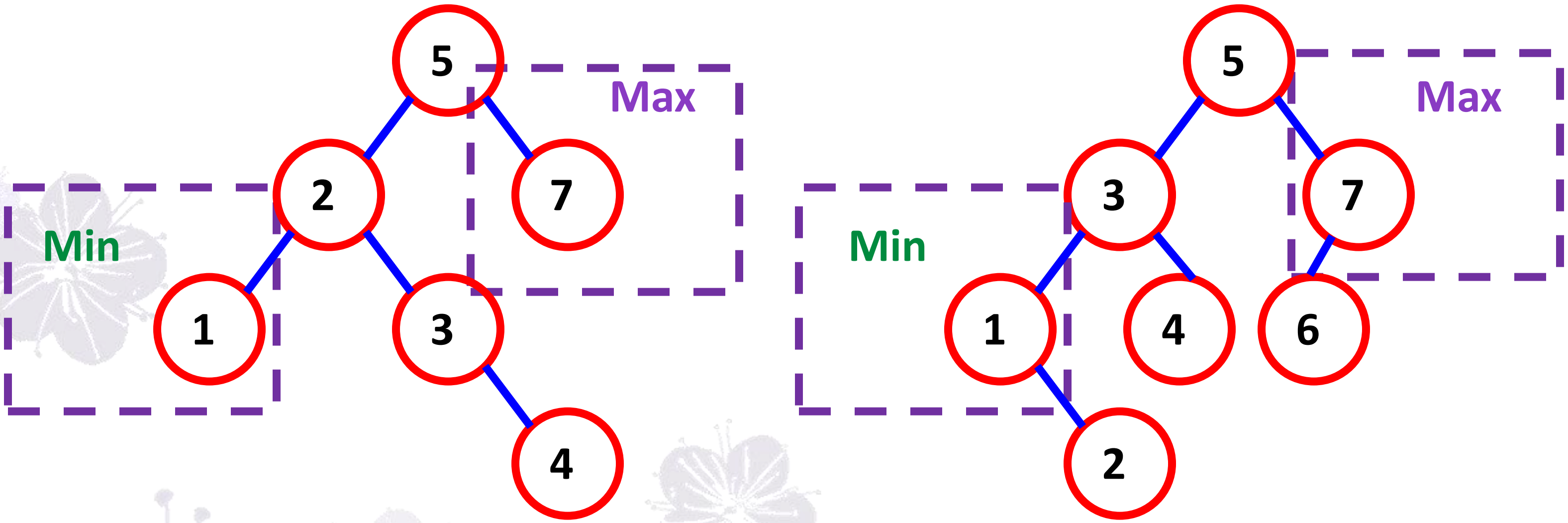


BST : Insert Codes

```
template < class K, class E >
void BST<K,E>::Insert(const pair<K,E>& thePair)
{ // Search for key "thePair.first", pp is the parent of p
  TreeNode<pair<K,E>>* p = root, *pp=0;
  while(p){
    pp = p;
    if(thePair.first < p->data.first)
      p = p->leftChild;
    else if(thePair.first > p->data.first)
      p = p->rightChild;
    else // Duplicate, update the value of element
    { p->data.second = thePair.second; return; }
  }
  // Perform the insertion
  p = new pair<K,E>(thePair);
  if(root) // tree is not empty
    if(thePair.first < pp->data.first) pp->leftChild = p;
    else pp->rightChild = p;
  else root = p;
}
```

BST : Delete

- **Min (Max)** element is at the **leftmost (rightmost)** of the tree



- Min or max are not always terminal nodes
- Min or max has ***at most one child***

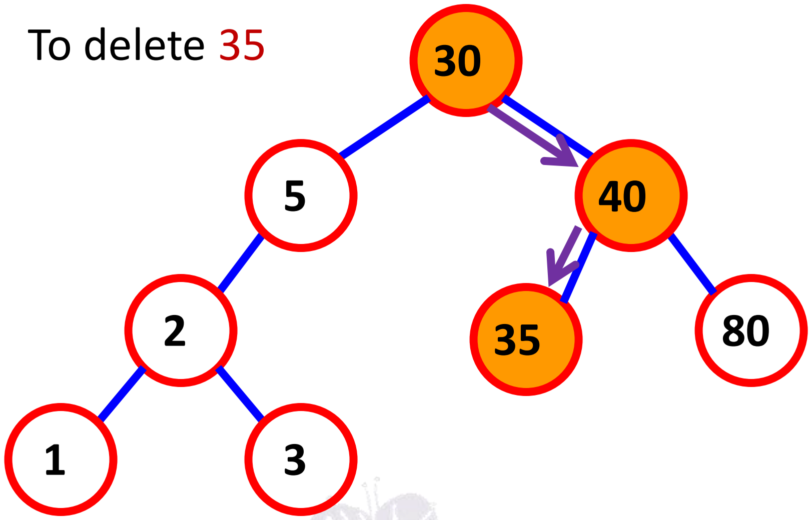
BST : Delete

1. To delete an element with key k
2. Search for the key k
3. If the search is successful, we have to deal three scenarios
   – The element is a **leaf** node
   – The element is a **non-leaf** node with **one child**
   – The element is a **non-leaf** node with **two children**

BST : Delete

- Scenario 1 : The element is a leaf node

To delete 35

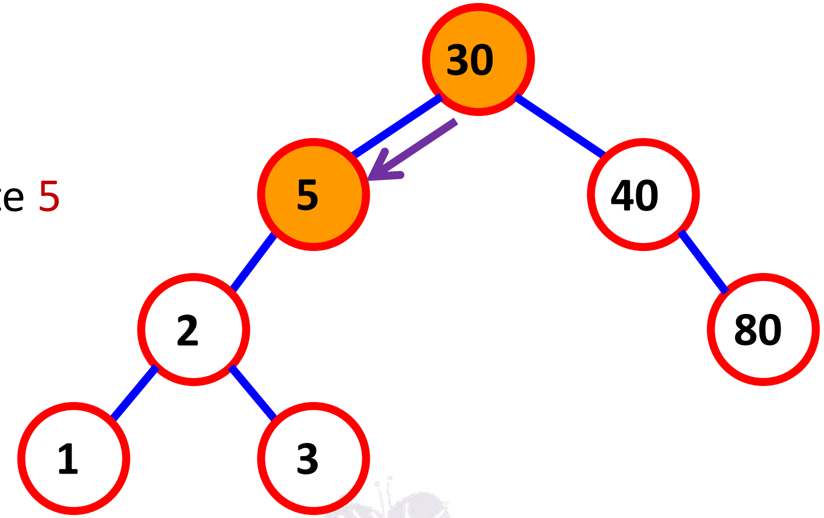


- The child field of parent node is set to NULL
- Dispose the node

BST : Delete

- Scenario 2 : The element is a non-leaf node with one child

To delete 5

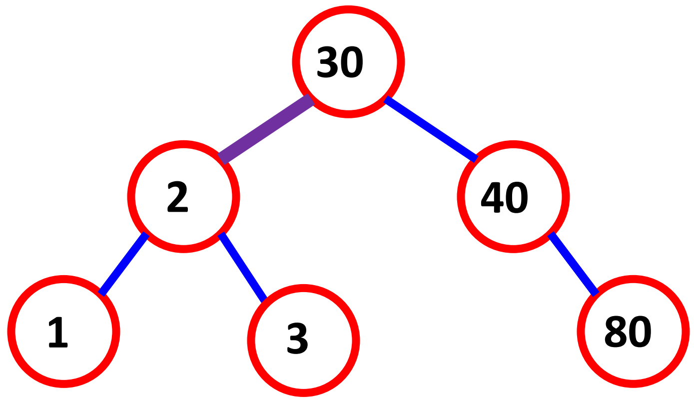


- Simply change the pointer from the parent node (i.e. node with key 30) to the single-child node (i.e. node with key 2)
- Dispose the node

BST : Delete

- Scenario 2 : The element is a non-leaf node with one child

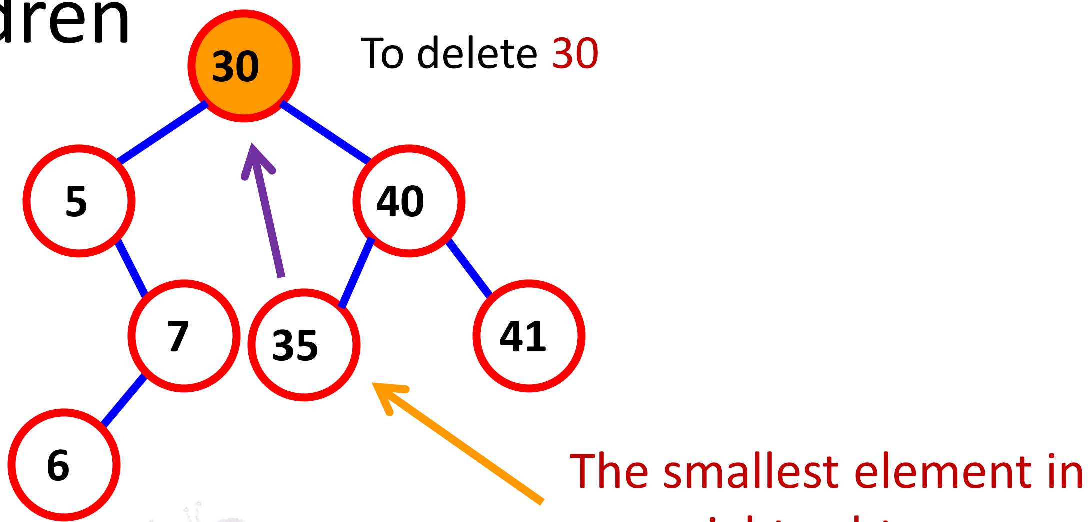
To delete 5



- Simply change the pointer from the parent node (i.e. node with key 30) to the single-child node (i.e. node with key 2)
- Dispose the node

BST : Delete

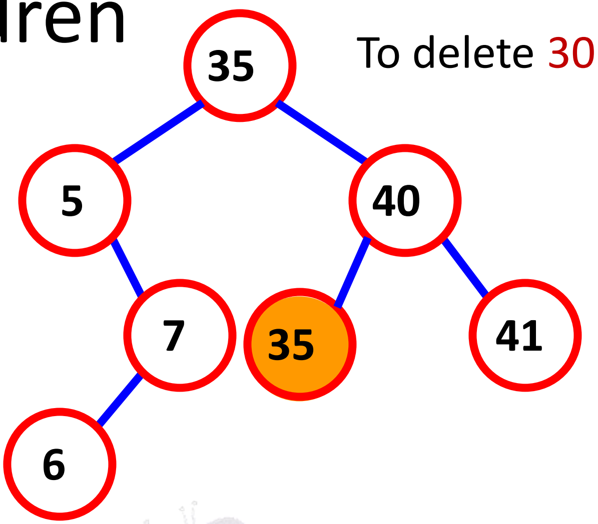
- Scenario 3 : The element is a non-leaf node with two children



- The deleted element is replaced by either
  - the **smallest** element in **right** subtree or
  - the **largest** element in **left** subtree

BST : Delete

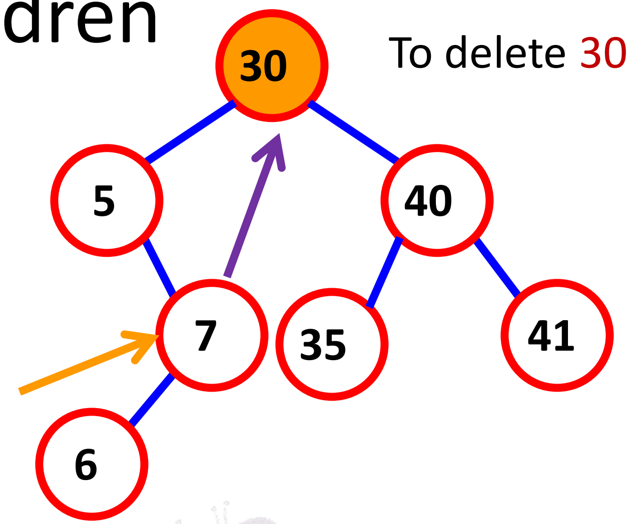
- Scenario 3 : The element is a non-leaf node with two children



- Delete the node
  - It is a leaf node -> apply scenario 1!

BST : Delete

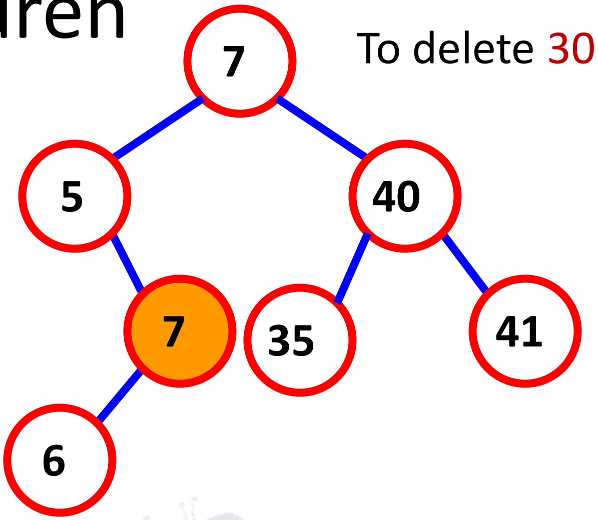
- Scenario 3 : The element is a non-leaf node with two children



- The deleted element is replaced by either
  – the **smallest** element in **right** subtree or
  – the **largest** element in **left** subtree

BST : Delete

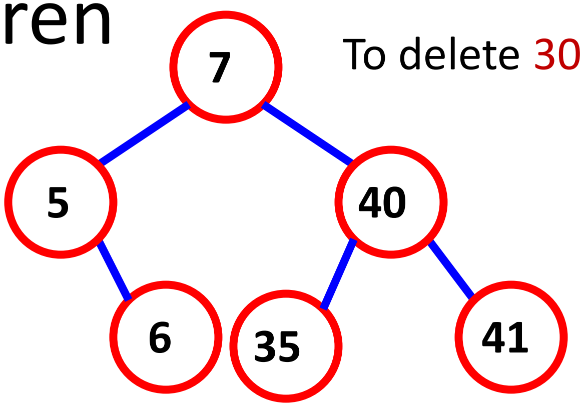
- Scenario 3 : The element is a non-leaf node with two children



- Delete the node
  - It is a non-leaf node with one child -> apply scenario 2!

BST : Delete

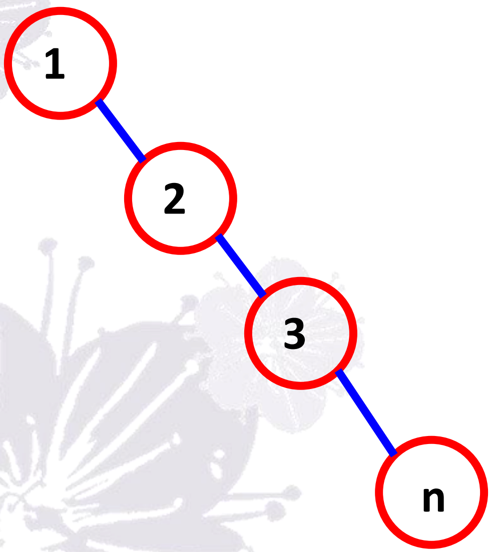
- Scenario 3 : The element is a non-leaf node with two children



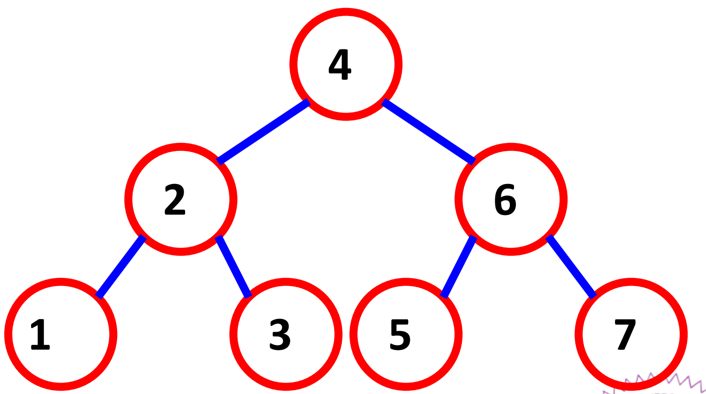
- Delete the node
  - It is a non-leaf node with one child -> apply scenario 2!

BST : Time Complexity

- Search, insertion, or deletion takes O(h)
- h = Height of a BST
- Worst case h=n
  - Insert keys 1, 2, 3, …



- Best case h=logn
  - Insert keys : 4, 2, 6, 1, 3, 5, 7



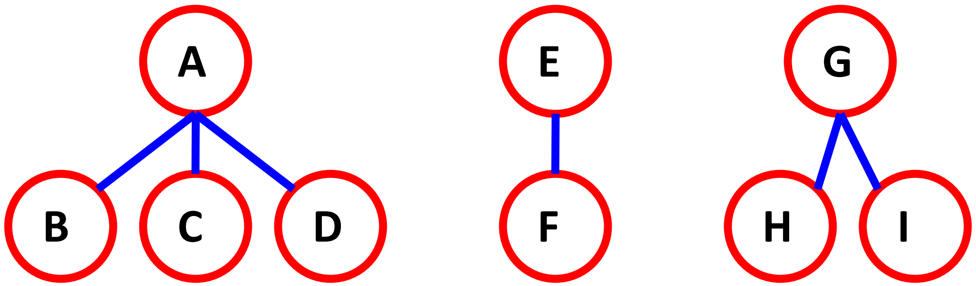
Self-Study Topics

- **Write pseudo codes of BST deletion**
- Selection trees
- AVL/Red-Black trees (Chapter 10)
  - Worst case height : O(logn)

FORESTS

Forests

- Definition : A forest is a set of n ≥ 0 disjoint trees.

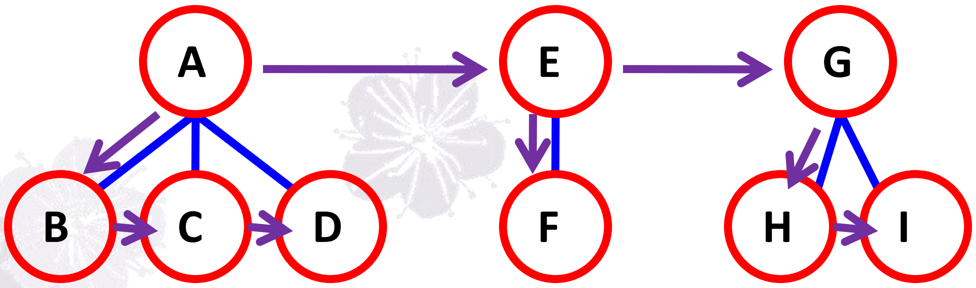


Three-tree forest

- Operations :
  - Transforming a forest to binary tree
  - Forest traversals

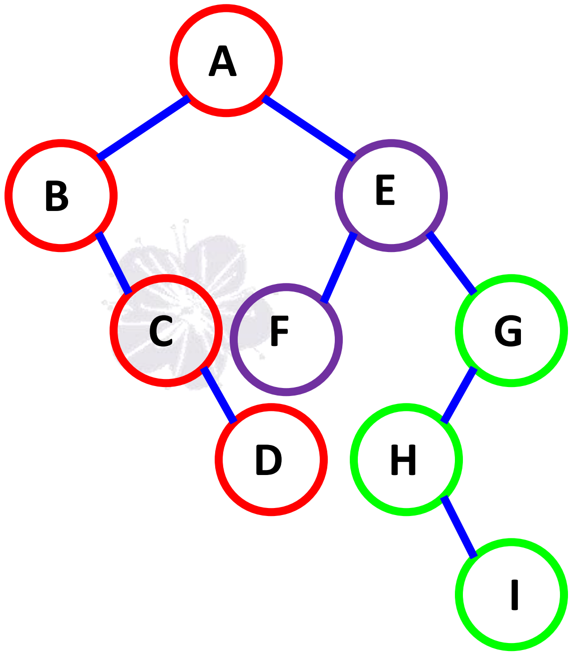
Transforming a Forest to Binary Tree

- Apply left child-right sibling approach
  - Convert each tree into binary tree
  - Connect two binary trees, T_1 and T_2, by setting the rightChild of root(T_1) to the root(T_2)



Transforming a Forest to Binary Tree

- Apply left child-right sibling approach
  - Convert each tree into binary tree
  - Connect two binary trees, T_1 and T_2, by setting the rightChild of root(T_1) to the root(T_2)

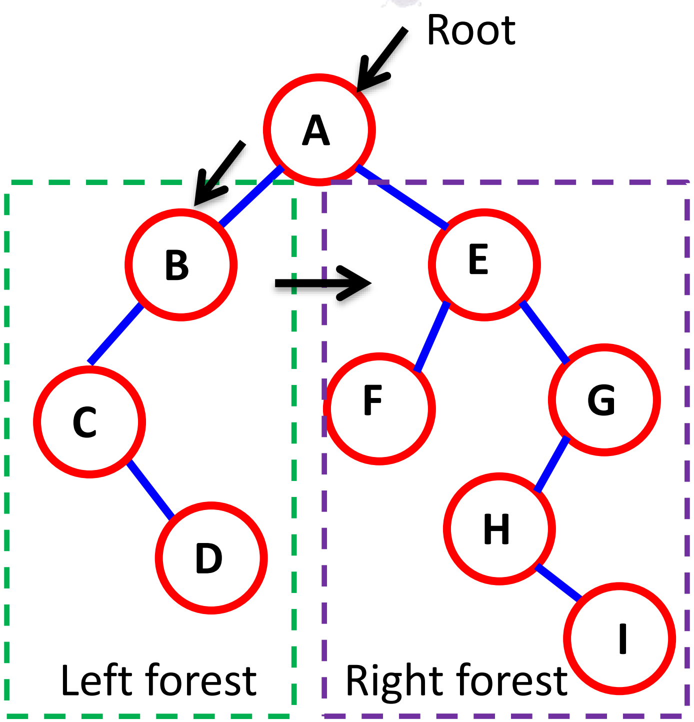


Forest Traversals

- Assume we have a forest **F** and corresponding binary tree **T**, then
- ***Preorder (inorder)*** traversal of **T** is equivalent to visiting the nodes of **F** in ***forest preorder (inorder)***

Forest Preorder Traversal

- Preorder traversal of binary tree
  - A B C D E F G H I
- Preorder traversal of forest
  - Root: A
  - Left forest: B C D
  - Right forest: E F G H I



Disjoint Sets

- Assume a set **S** of **n** integers **{0, 1, 2,…, n-1}** is divided into several subsets $\mathbf{S_1, S_2, \ldots, S_k}$ and $\mathbf{S_i} \cap \mathbf{S_j} = \boldsymbol{\phi}$ for any **i, j ϵ { 1, … , k } and i ≠ j**

- Operations:
  – Disjoint set union : **Union(S_i, S_j)**
    - $S_i = S_i \cup S_j$ or $S_j = S_i \cup S_j$
  – Find the set containing element x : **Find(x)**

Disjoint Sets : Example

- Set
  - $S = \{ 0,1, 2, 3, 4, 5 \}$
- Disjoint subsets
  - $S_1 = \{ 0, 2, 3 \}$
  - $S_2 = \{ 1 \}$
  - $S_3 = \{ 4, 5 \}$
- $\text{Union}(S_1, S_2) = \{ 0, 1, 2, 3 \}$
- $\text{Find}(5) = 3$

DS: Array Representation

- $S = \{ 0, 1, 2, 3, 4, 5 \}$ with subsets
  - $S_1 = \{ 0, 2, 3 \}$, $S_2 = \{ 1 \}$ and $S_3 = \{ 4, 5 \}$
- Using a **sequential mapping array** where index represents set members and array value indicates **set name**

Set name →	1	2	1	1	3	3
Set member →	S[0]	S[1]	S[2]	S[3]	S[4]	S[5]

DS Operation: Find(x)

Set name →	1	2	1	1	3	3
Set member →	S[0]	S[1]	S[2]	S[3]	S[4]	S[5]

- Find the set which contains element x is easy
  - Find(5) = S[5] = set 3
    Find(3) = S[3] = set 1
  - Complexity = O(1)

DS Operation: Union(S_i, S_j)

Set name →	1	2	1	1	3	3
Set member →	S[0]	S[1]	S[2]	S[3]	S[4]	S[5]

- Assume we always merge the 2nd set to 1st set
  $S_i = S_i \cup S_j$
- Scan the array and **set S[k] to i if S[k]==j**
  - S_2=Union(S_2, S_3)

Set name →	1	2	1	1	2	2
Set member →	S[0]	S[1]	S[2]	S[3]	S[4]	S[5]

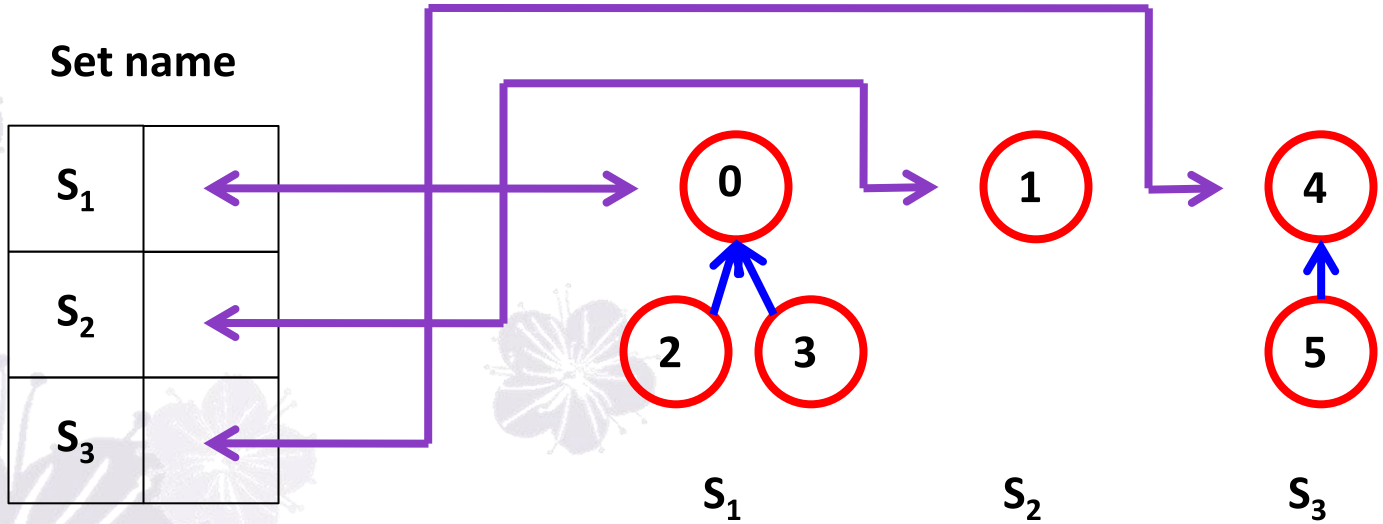
DS Time Complexity

- $S = \{ 0, 1, 2, \ldots , n-1 \}$
  - $S_1 = \{ 0 \}$, $S_2 = \{ 1 \}$, $S_3 = \{ 2 \}$, … , $S_n = \{ n-1 \}$
- Perform a sequence Union
  - $Union(S_2, S_1), Union(S_3, S_2), \ldots, Union(S_n, S_{n-1})$
  - $(n-1)*O(n) = O(n^2)$
- Followed by a sequence of Find
  - Find(0), Find(1), …, Find(n-1)
  - $n*O(1)=O(n)$
- Total time complexity = $O(n^2)$

DS: Tree Representation

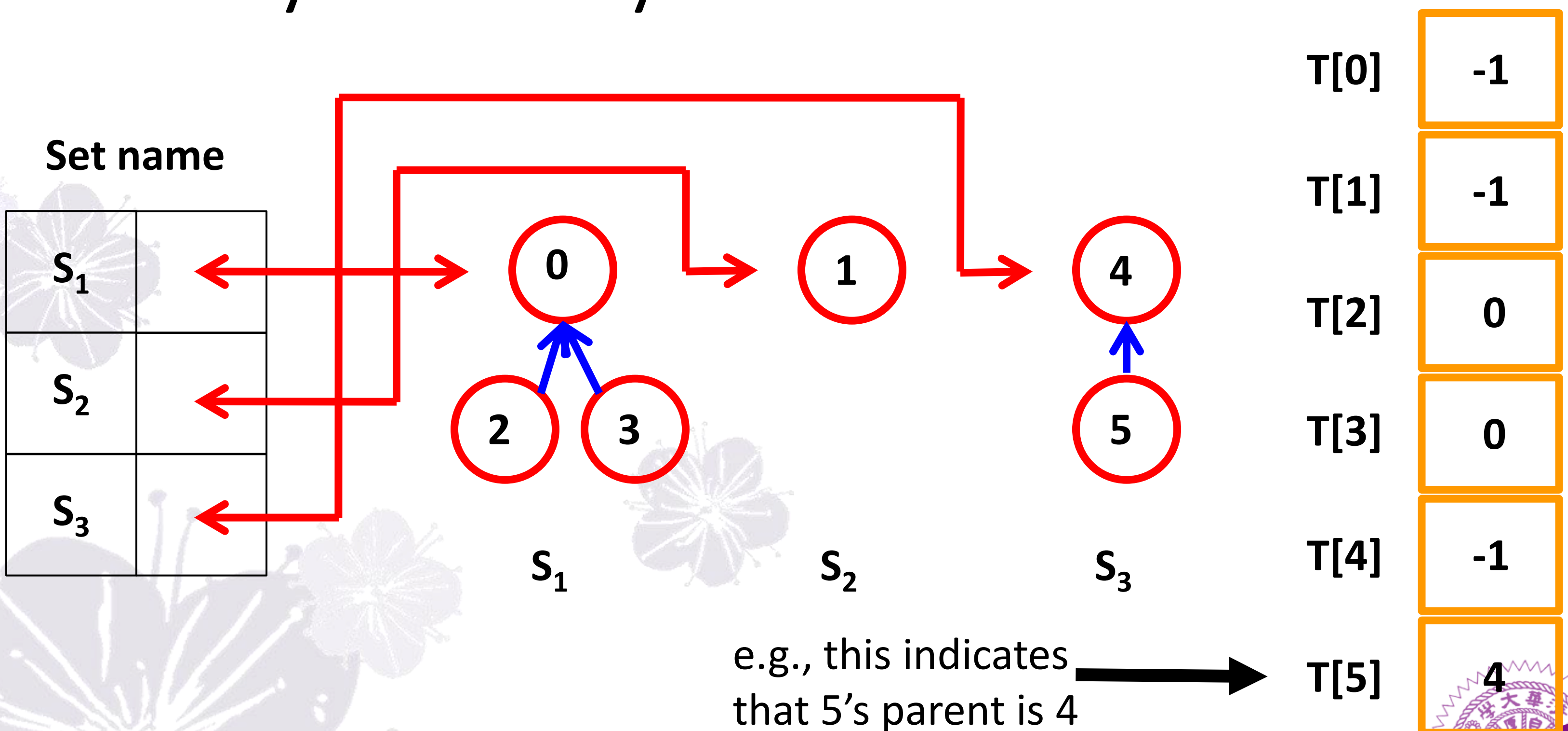
- Link elements of a subset to form a tree
  - Link children to root
  - Link root to set name

$S = \{ 0, 1, 2, 3, 4, 5 \}$ with subsets
$S_1 = \{ 0, 2, 3 \}$, $S_2 = \{ 1 \}$ and $S_3 = \{ 4, 5 \}$



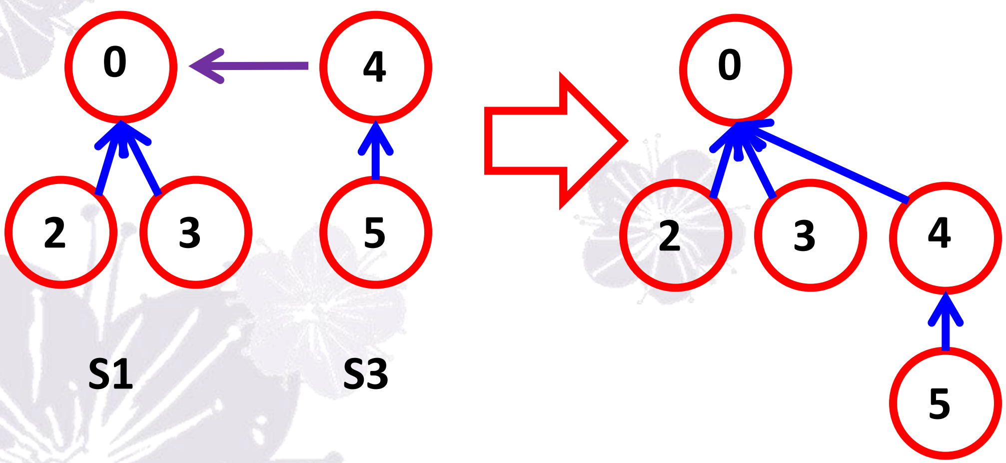
DS: Tree Representation

- Use an array to store the tree
- Identify the set by the root of the tree



DS Operation: Union(S_i, S_j)

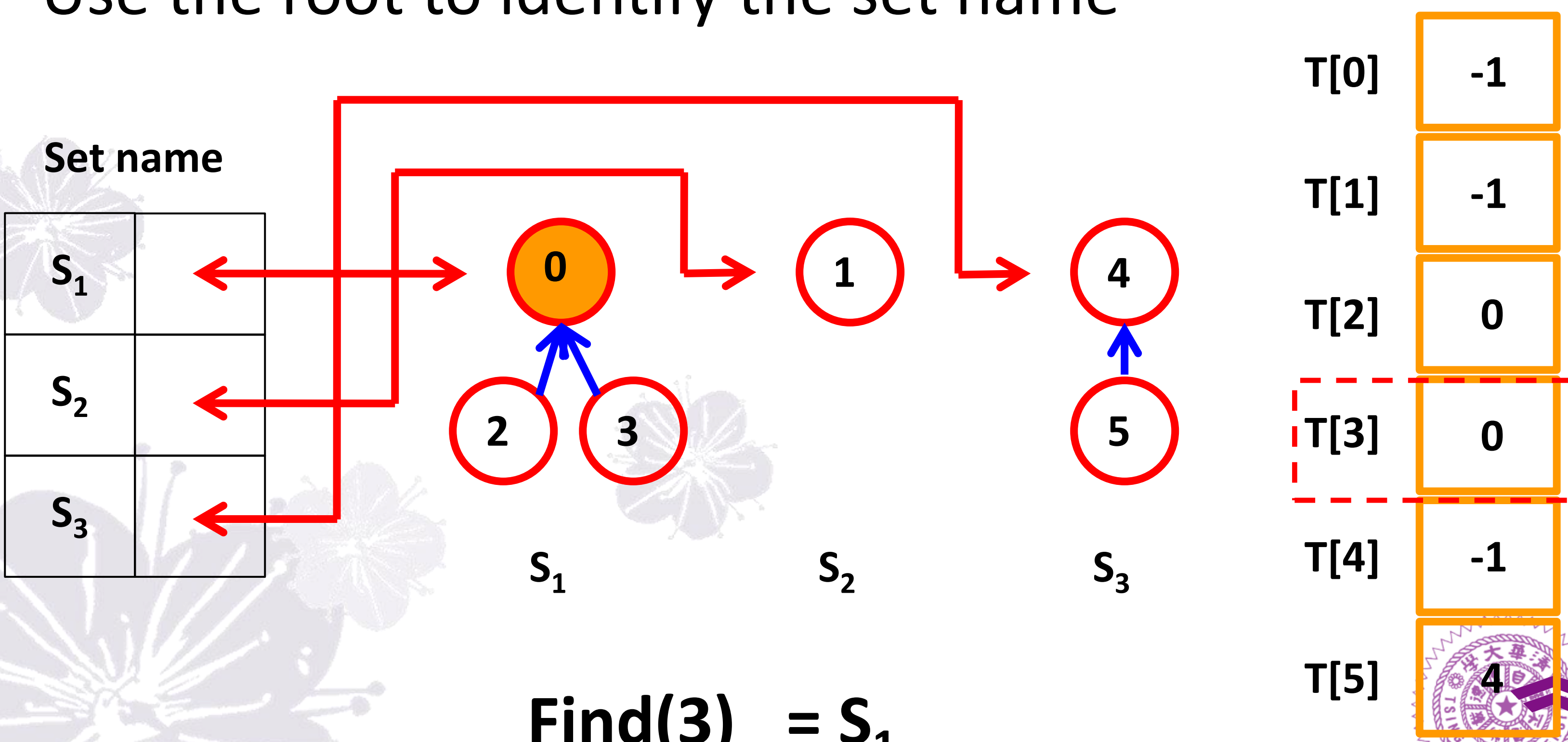
- Set the parent field of one of the root to the other root
  - S_1=Union(S_1, S_3)
  - Time complexity : O(1)



T[0]	-1
T[1]	-1
T[2]	0
T[3]	0
T[4]	0
T[5]	4

DS Operation: Find(x)

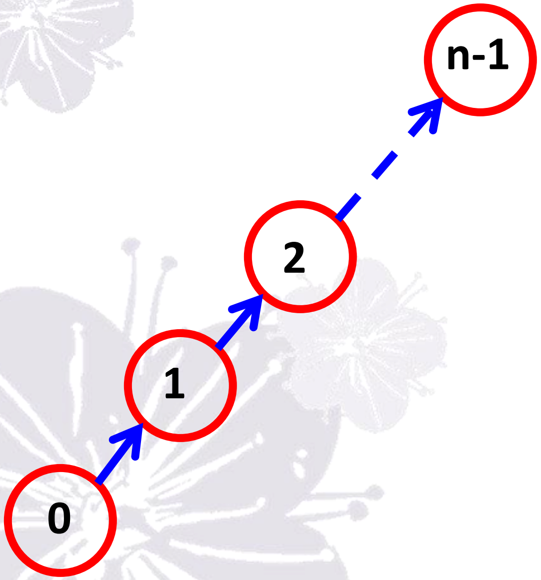
- Following the index starting at x and tracing the tree structure until reaching a node with parent value = -1
- Use the root to identify the set name



Find(3) = S_1

DS Time Complexity

- $S = \{ 0, 1, 2, \ldots, n\text{-}1 \}$
  - $S_1 = \{ 0 \}$, $S_2 = \{ 1 \}$, $S_3 = \{ 2 \}$, … , $S_n = \{ n\text{-}1 \}$
- Perform a sequence Union
  - Union(S_2, S_1), Union(S_3, S_2), …, Union(S_n, S_{n-1})

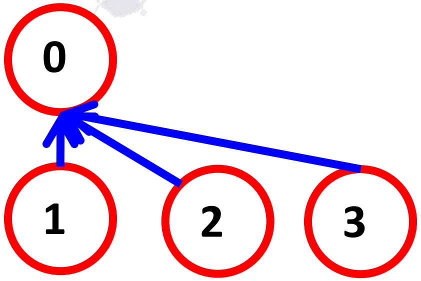


Followed by a sequence of Find
Find(0), Find(1), …, Find(n-1)

Time Complexity = $\sum_{i=1}^{n} i = O(n^2)$

Improved Union(S_i, S_j)

- Do not always merge two sets into the first set
- Adopt a ***Weighting rule*** to union operation
  - $S_i = S_i \cup S_j$, if $| S_i | >= | S_j |$
  - $S_j = S_i \cup S_j$, if $| S_i | < | S_j |$
- $S = \{ 0, 1, 2, \ldots , n \}$
  - $S_1 = \{ 0 \}$, $S_2 = \{ 1 \}$, $S_3 = \{ 2 \}$, … , $S_n = \{ n-1 \}$
  - Union (1, 2)->Union (1, 3)->Union (1, 4)



Maximum Tree Height

- Lemma 5.5
  - Let T be a tree with m nodes created by a sequence of weighting unions.
    The height of T is no greater than $\lfloor \log_2 m \rfloor + 1$

- Proof
  - The longest length is the path that is increased by 1 in every union operation
  - You may check the detailed proof in page 310

Maximum Tree Height (Proof)

Let T be a tree with m nodes created by a sequence of weighting unions.
The height of T is no greater than $\lfloor \log_2 m \rfloor + 1$

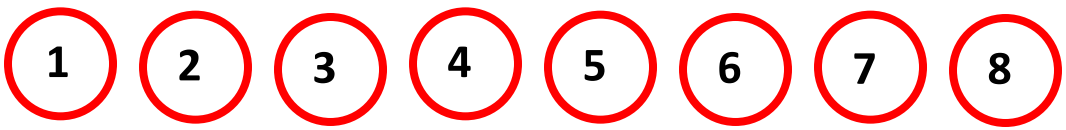
- Lemma 5.5
  - Proved with induction
  - Clearly true for m=1
  - Assume true for all trees with i nodes, i<=m-1. Now we prove that it is also true for i=m.
  - Let T be a tree with m nodes created by WeightedUnion. Consider the last union operation, say union(k,j). Let a be the number of nodes in tree j and m-a the number in tree k. WLOG, assume 1<=a<=m/2.
  -

Let T be a tree with m nodes created by a sequence of weighting unions.
The height of T is no greater than $\lfloor \log_2 m \rfloor + 1$

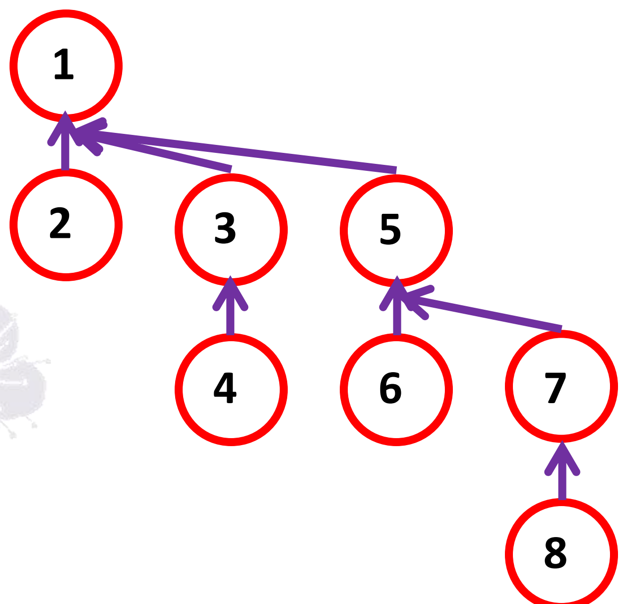
- Then the height of T is either i) the same as k, or ii) is one more than that of j. If i) holds, the height of T is $\leq \lfloor log_2(m-a) \rfloor + 1 \leq \lfloor log_2(m) \rfloor + 1$ $(by\ assumption\ of\ induction)$
- If ii) holds, the height of T is $\leq \lfloor log_2 a \rfloor + 2 \leq \left\lfloor log_2 \frac{m}{2} \right\rfloor + 2 \leq \lfloor log_2(m) \rfloor + 1.$

Time Complexity

- The following sequence of unions produces the height of log n



- Union(1, 2)
- Union(3, 4)
- Union(5, 6)
- Union(7, 8)
- Union(1, 3)
- Union(5, 7)
- Union(1, 5)



For (n-1) unions and n find => O(n log n)