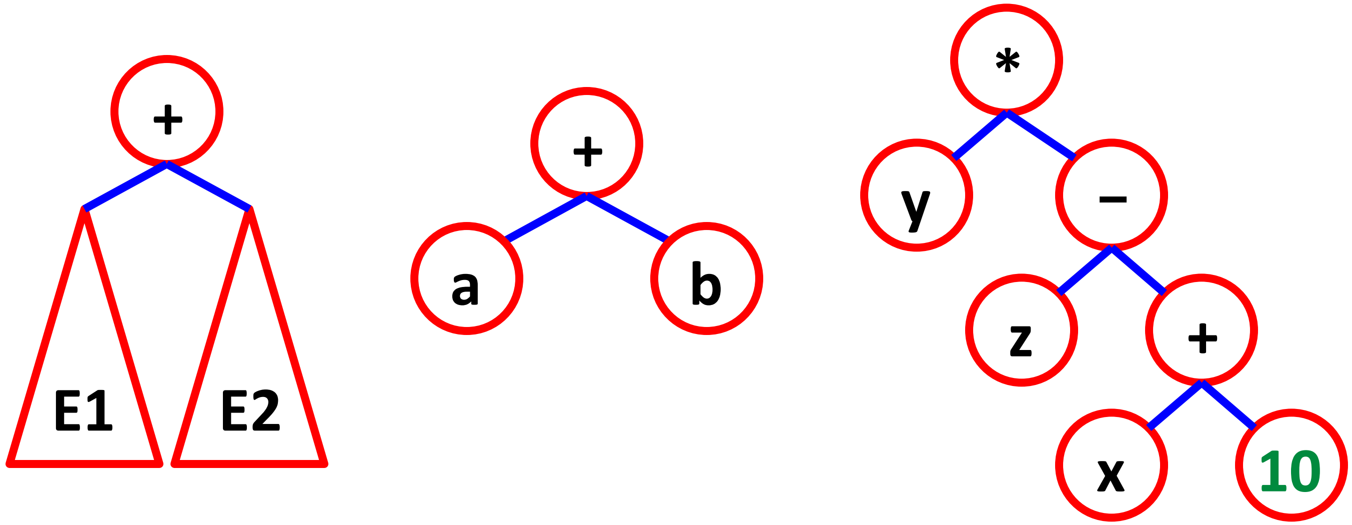
CS 2351 Data Structures

Trees (II)

Prof. Chung-Ta King
Department of Computer Science
National Tsing Hua University

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))
Preorder	+ E1 E2	+ a b	* y – z + z 10
Postorder	E1 E2 +	a b +	y z x 10 + – *

Infix notation

Prefix notation

Postfix notation

Outline

- Heap (Sec. 5.6)
  - Priority queues, max heap
- Binary search trees (Sec. 5.7)

Priority Seat

Passengers are ordered according to a certain criteria, not just arrival time

Who Is Next in Line?

- Who has the next highest priority?
  - We care less who are the third, fourth, …, in line

Priority Queue

- A queue that orders the elements by importance or priority
- The element to be processed/deleted is the one with the **highest** (or **lowest**) priority
- Operations
  - Get the element with the max/min priority
  - Insert an element to the priority queue
  - Delete an element with the max/min priority
  - Don’t care which is the *n*-th highest priority

ADT of 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 the max priority
    void Pop();
private:
    // Data representation here
};
```

Implementing Priority Queue

- Unsorted linear list
  - Array, chain, … → no ordering
- Sorted linear list
  - Sorted array, sorted chain, … → total ordering
- Heap → partial ordering

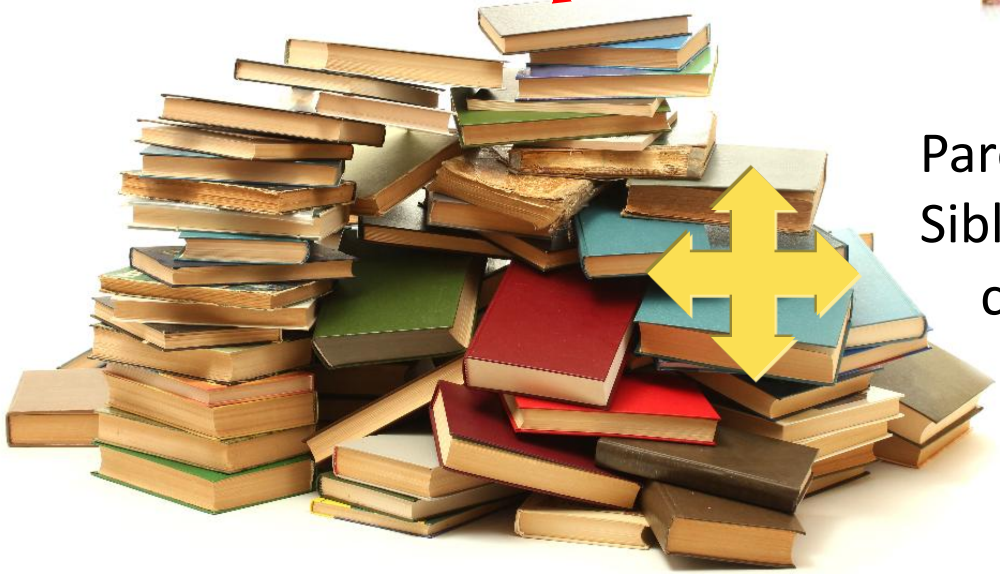
	Top() (Search)	Push() (Insert)	Pop() (Delete)
Unsorted linear list	O(n)	O(1)	O(n)
Sorted linear list	O(1)	O(n)	O(1)
Heap	O(1)	O(logn)	O(logn)

Heap

- *A ~~disorganized~~ pile of things*

partially ordered

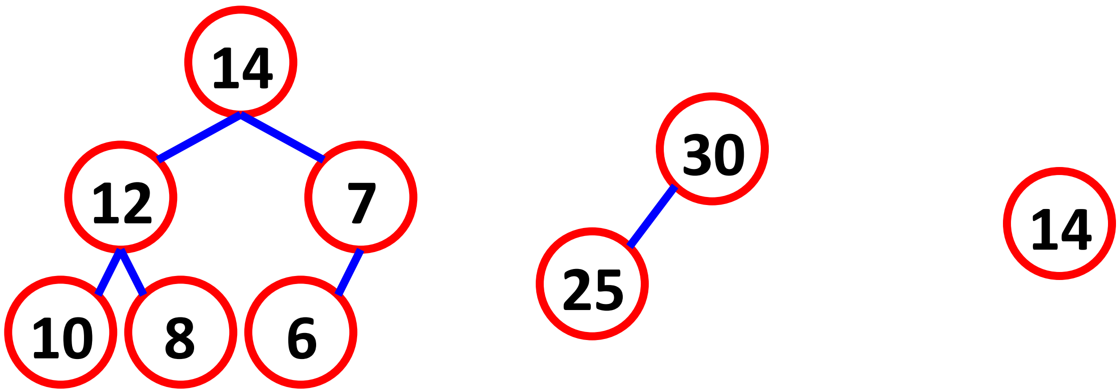
The one at the top
is important



Parent-child ordering important
Sibling ordering unimportant
c.f. queue

Max/Min Heap

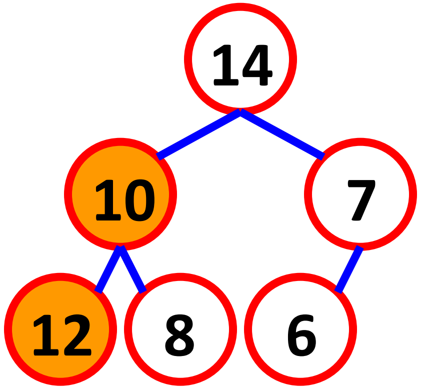
- A ***max (min) tree*** is a tree in which the key value in each node is ***no smaller*** (***larger***) than the key values in its children (if any)
- A ***max (min) heap*** is a ***complete binary tree*** that is also a ***max (min) tree*** → **root is the max (min)**



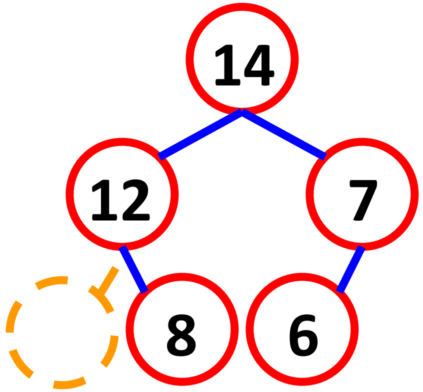
Max Heap Max Heap Max/Min Heap

Max/Min Heap

- 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*** → **root is the max (min)**



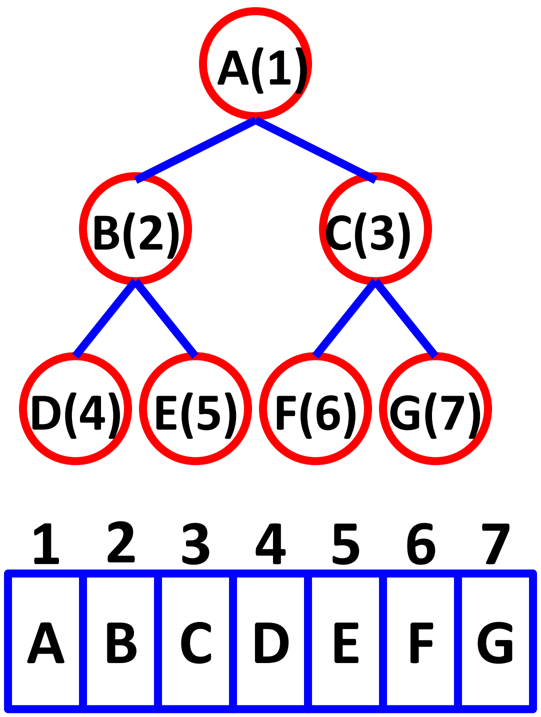
Not a heap
(12 > 10)



Not a heap
(Not a complete binary tree)

Array Representation of Max Heap

- Since the heap is a complete binary tree, we could adopt "**Array Representation**" as 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, i has no left child
  - **rightChild(i) = 2i+1** if 2i+1 ≤ n;
    if 2i+1 > n, i has no right child



ADT of 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 the max priority
     void Pop();
private:
     T* heap           // Element array
     int heapSize; // # of elements
     int capacity; // size of the array "heap"
};
```

Max Heap in C++

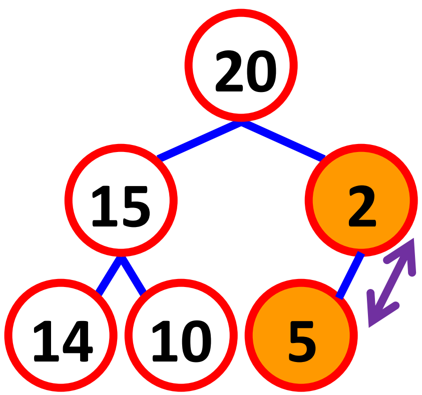
```
template <class T> class MaxPQ;
template <class T> class MaxHeap;
template <class T> class Element {
friend class MaxPQ<T>;
friend class MaxHeap<T>;
public:
    Element(T k = 0) : key(k) {};
private:
    T key;
};
template <class T> class MaxPQ {
public:
  virtual Element<T> *Top() = 0;
  virtual void Push(const Element<T>&) = 0;
  virtual Element<T>* Pop(Element<T>&) = 0;
};
```

Max Heap in C++

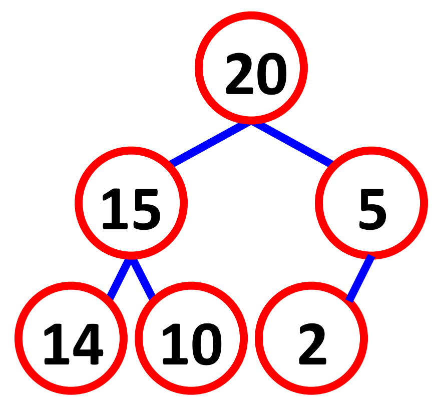
```
template <class T> class MaxHeap : public MaxPQ<T> {
public:
  MaxHeap(int sz = defaultHeapSize) {
    capacity = sz; heapSize = 0;
    heap = new Element<T> [capacity + 1]; };
  Element<T> *Top() {return &heap[1];}
  void Push(const Element<T>& x);
  Element<T> *Pop(Element<T>&);
private:
  Element<T> *heap;
  int heapSize; // current size of MaxHeap
  int capacity; // Maximum allowable size of MaxHeap
  void HeapEmpty(){ cout << "Heap Empty" << "\n";};
  void HeapFull(){ cout << "Heap Full";};
};
```

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 the two nodes



1	2	3	4	5	6	7
20	15	2	14	10	5	-



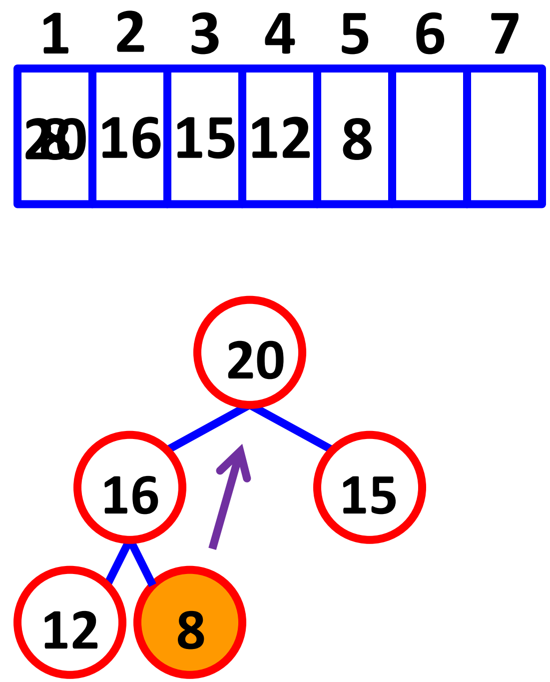
Max Heap: Insert

```
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 points to parent
  }
  heap[currentNode]=e;
}
```

Time complexity: Visit at most the height of the tree → O(log n)

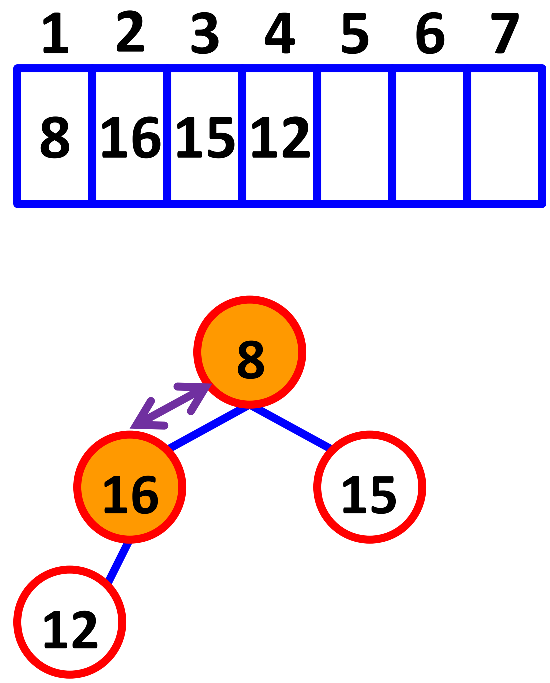
Max Heap: Delete

1. Always delete the root
2. Move the last element to the root (maintain a complete binary tree)



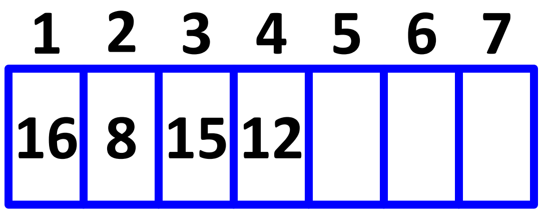
Max Heap: Delete

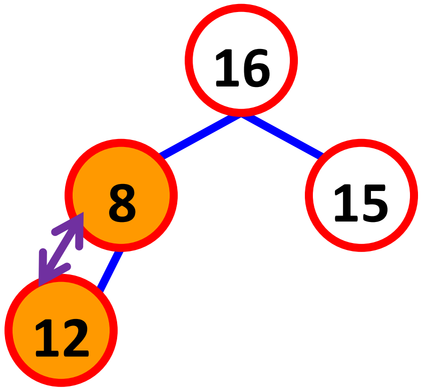
1. Always delete the root
2. Move the last element to the root (maintain a complete binary tree)
3. Swap with larger and largest child (if any)



Max Heap: Delete

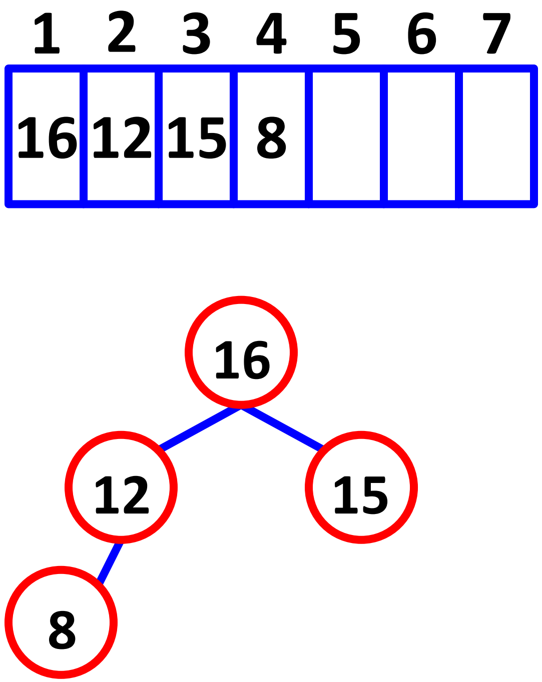
1. Always delete the root
2. Move the last element to the root (maintain a complete binary tree)
3. Swap with larger and largest child (if any)
4. Continue step 3 until the max heap is maintained (*trickle down*)





Max Heap: Delete

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



Max Heap: Delete

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

Outline

- Heap (Sec. 5.6)
  - Priority queues, max heap
- Binary search trees (Sec. 5.7)

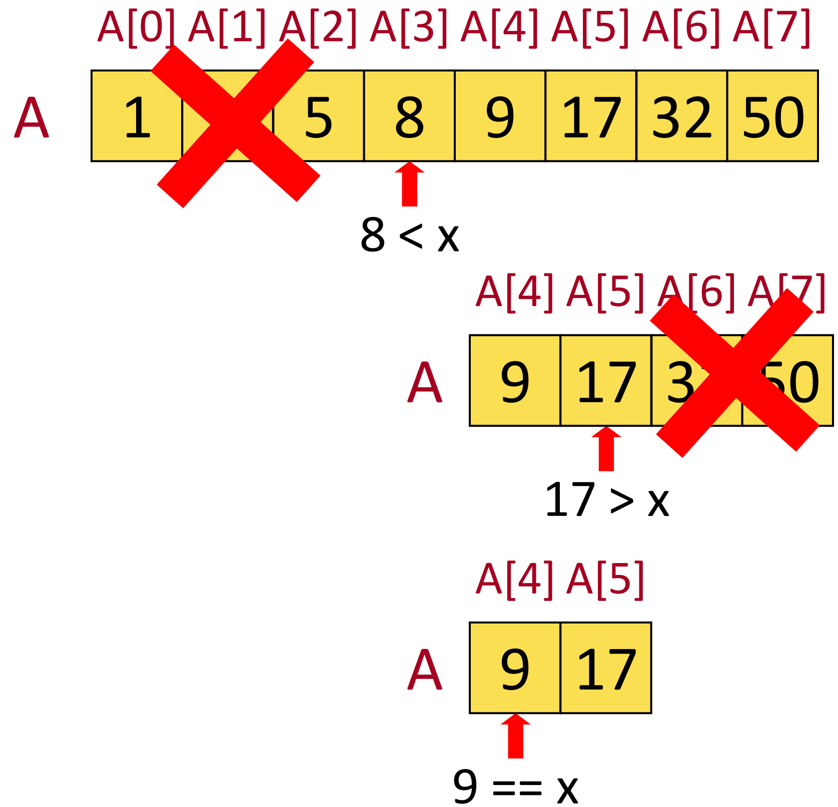
Recall Binary Search through Sorted Array

- Search for x=9 in array A[0], ..., A[7]:

1st iteration:

2nd iteration:

3rd iteration:

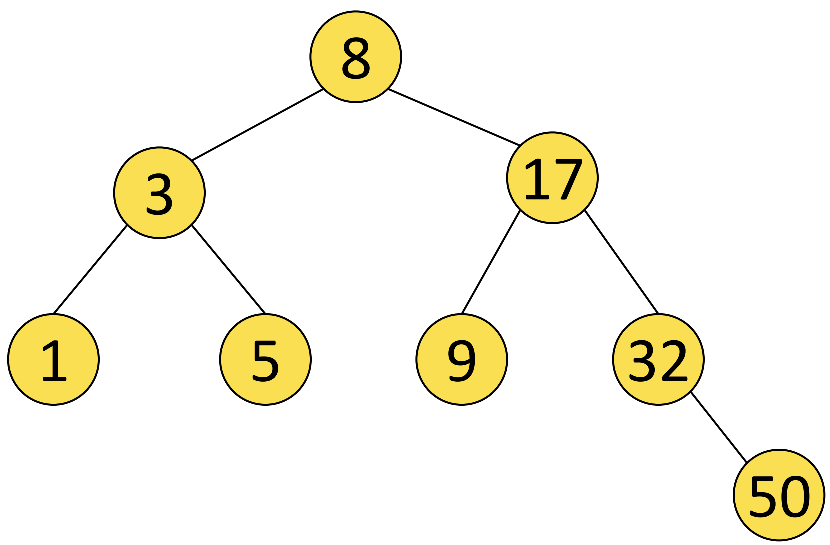


Search in O(log n) time, but insertion/deletion in ? time

How to Improve on Insertion/Deletion?

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
A	1	3	5	8	9	17	32	50

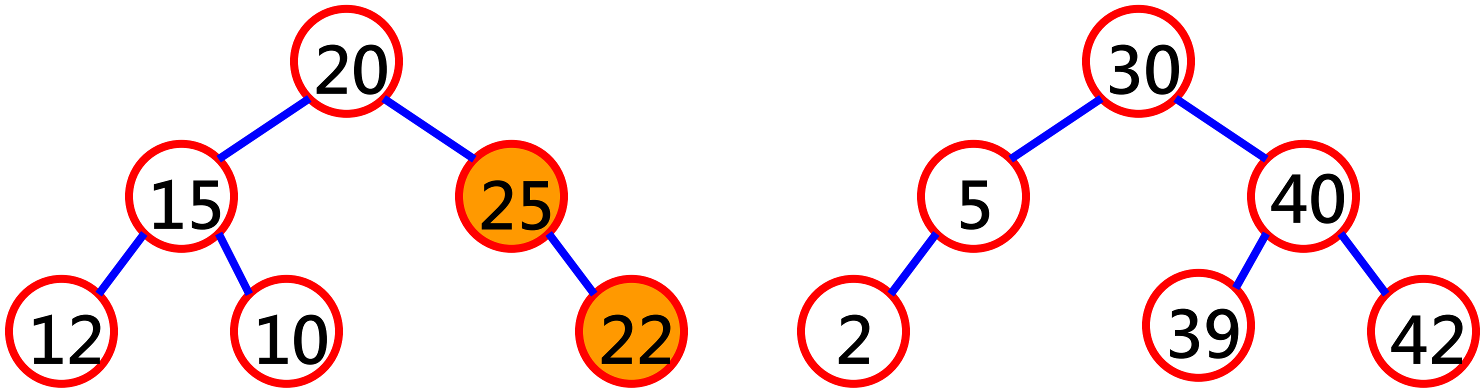
- **Use a tree!**



Binary Search Tree

- A ***binary search tree (BST)*** is a binary tree that:
  - Every element has a a (*key*, *value*) pair 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



NO! **YES!**

Inorder traversal?

Inorder traversal of a BST will result in a sorted list

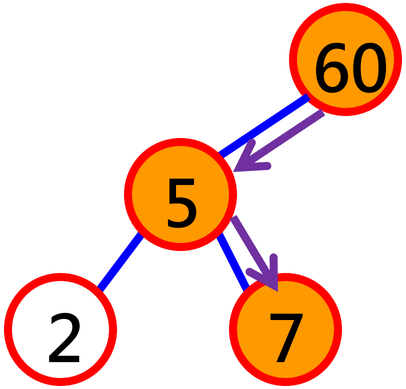
BST: Operations

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

BST: Search an Element

Search for key 7

- Start from root
- Compare the key with root
  - ‘<’ search the left subtree
  - ‘>’ search the right subtree
- Repeat step 3 until the key is found or a leaf is visited



BST: Recursive Search

```
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 found, return its pointer; 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

Can you write a non-recursive version?

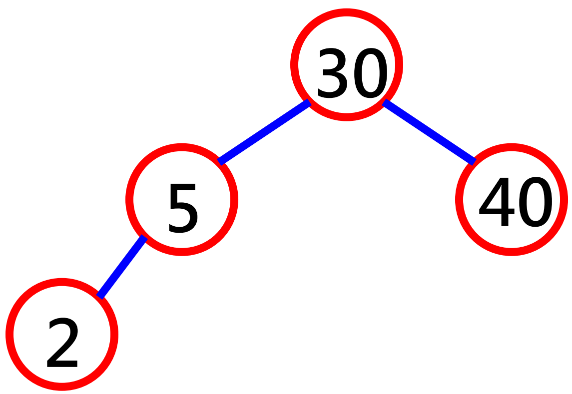
std::pair in STL

- A struct that provides for the ability to treat two objects as a single object
  - **`pair<T1,T2>`** is a heterogeneous pair: it holds one object of type T1 and one of type T2
  - The individual values can be accessed through its public members **`first`** and **`second`**
- Example:

```
pair<bool, double> result = foo();
if (result.first)
  do_something_more(result.second);
pair <int,char> element1(30,'x');
```

Can Also Search by Rank

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



Inorder traversal: 2 → 5 → 30 → 40

Rank: 1 2 3 4

Thus, the r^th smallest element is the node with rank r

BST: Search by Rank

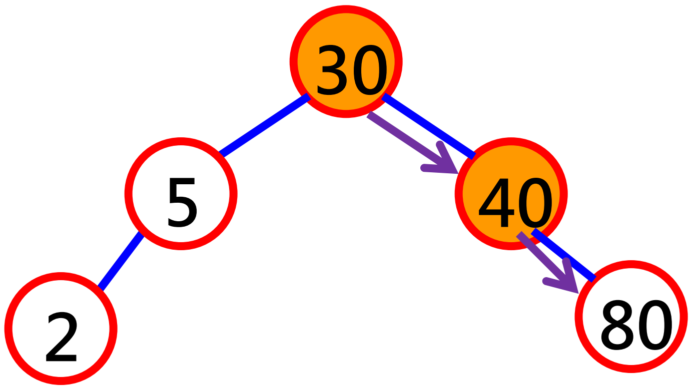
- For each node, we store an additional data “leftSize”, which is 1 + (# of nodes in the left subtree)

```
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;
}
```

BST: Insert

To insert an element with key 80

- First we search for the existence of the element
- If the search is unsuccessful, then the element is inserted at the point the search terminates

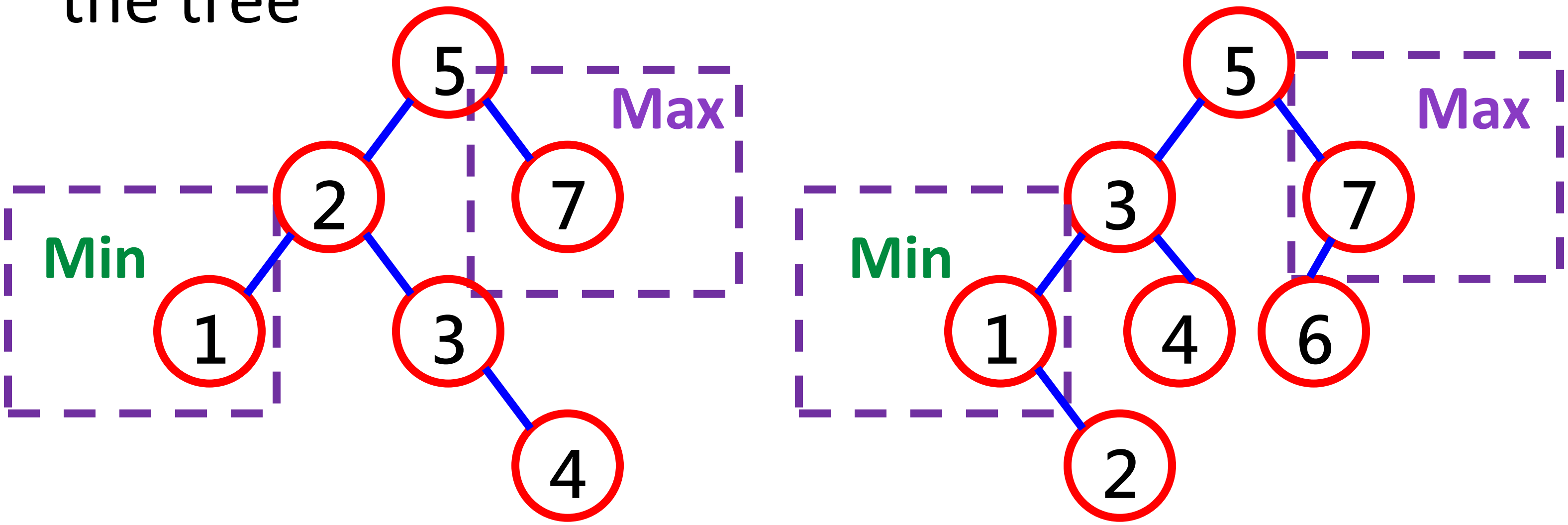


BST: Insert

```
template <class K, class E>
void BST<K,E>::Insert(const pair<K,E>& thePair)
{ // Search for key "thePair.first", pp is 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

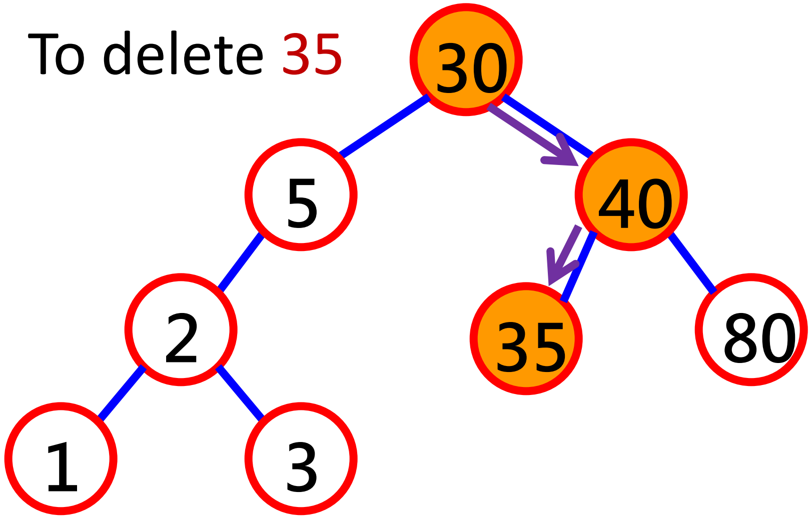
BST: Delete

To delete an element with key k

- Search for the key k
- If the search is successful, we have to deal with 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

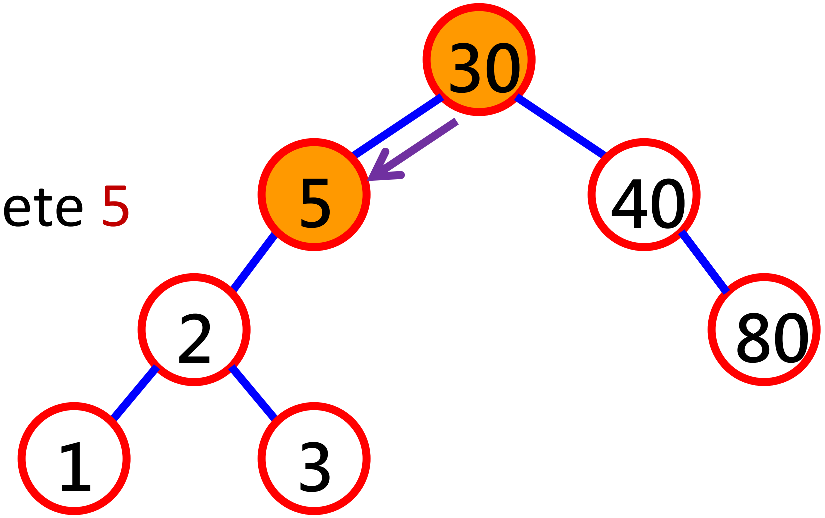


- The child field of the parent node is set to NULL
- Dispose the node

BST: Delete

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

To delete 5

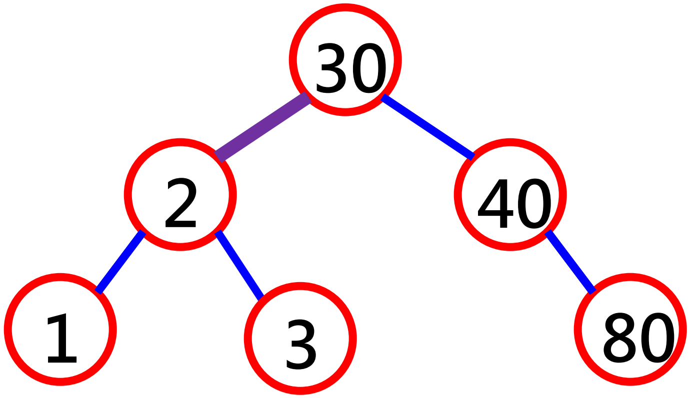


- Simply change the pointer from parent node (node with key 30) to single-child node (node with key 2)
- Dispose the node 5

BST: Delete

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

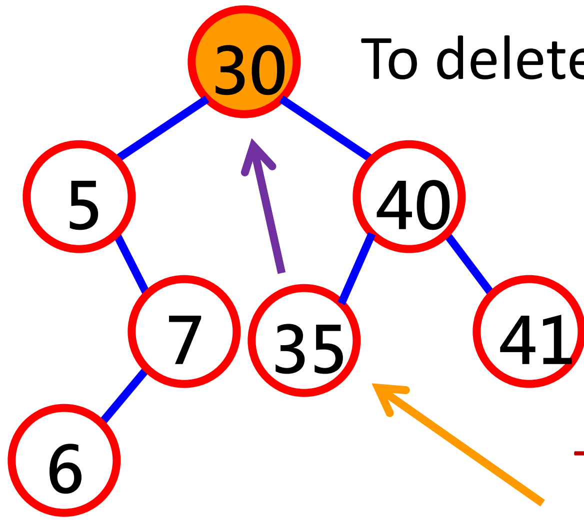
To delete 5



- Simply change the pointer from parent node (node with key 30) to single-child node (node with key 2)
- Dispose the node 5

BST: Delete

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



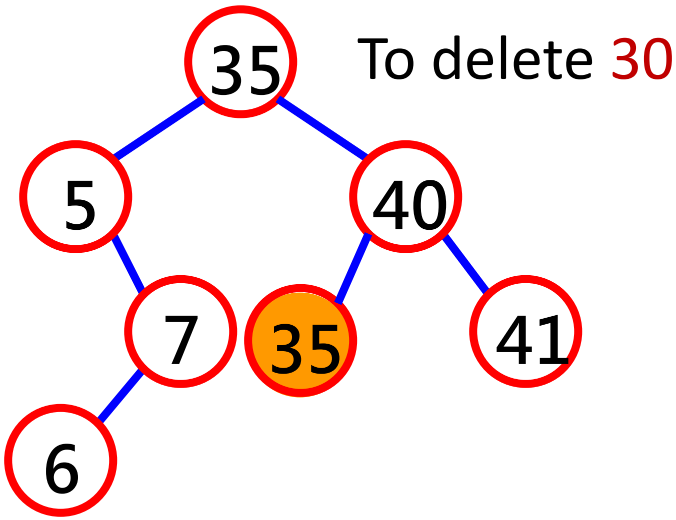
To delete 30

The smallest element in right subtree

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

BST: Delete

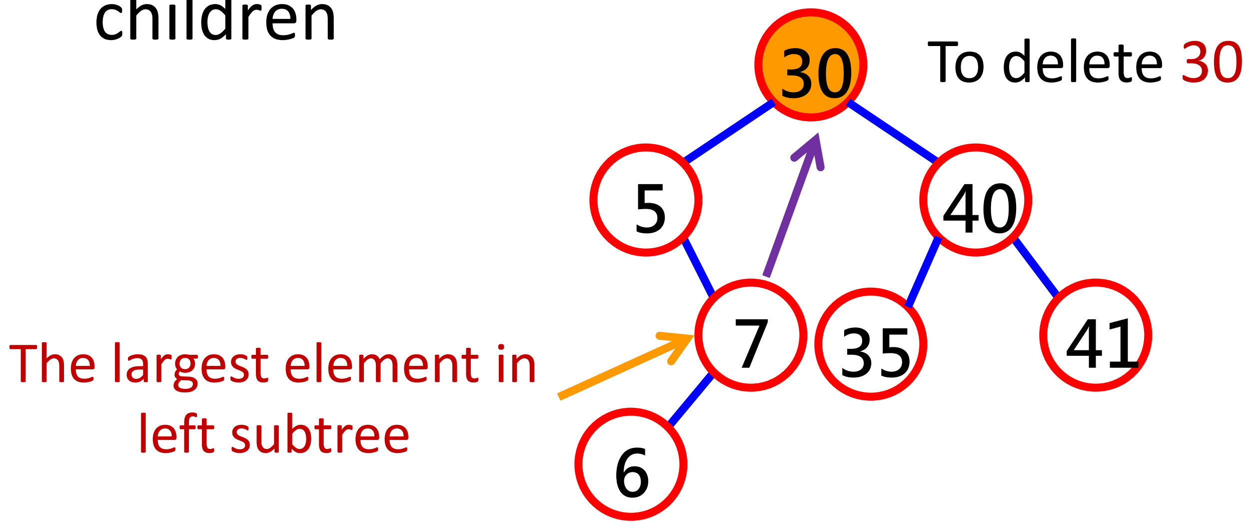
- Scenario 3: The element is a non-leaf node with two children



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

BST: Delete

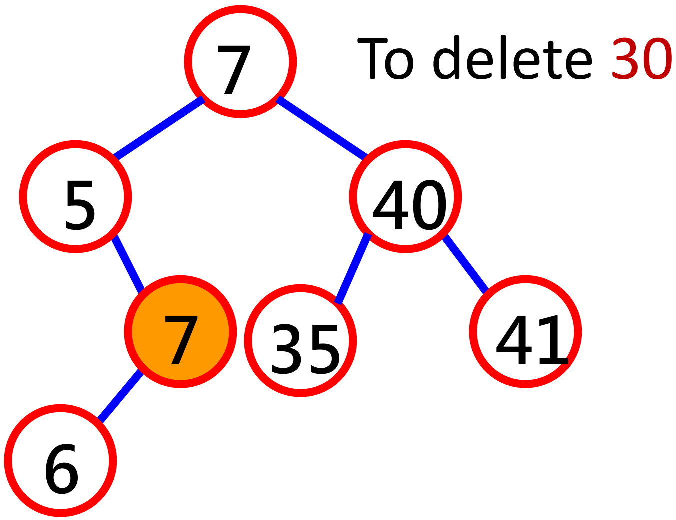
- Scenario 3: The element is a non-leaf node with two children



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

BST: Delete

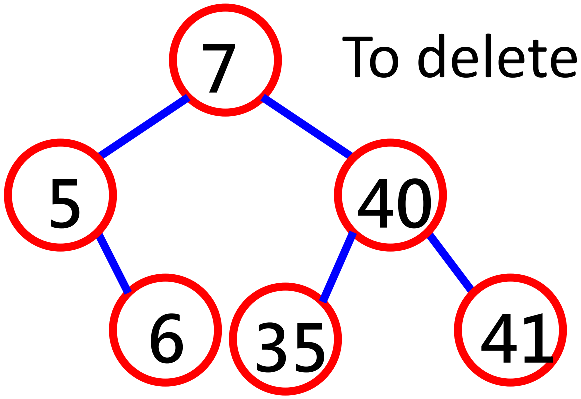
- Scenario 3: The element is a non-leaf node with two children



- 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



To delete 30

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

BST: Time Complexity

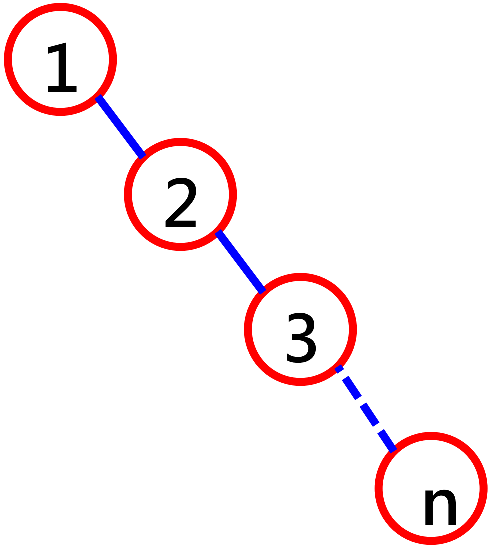
- Search, insertion, or deletion takes O(*h*)
- h = Height of a BST

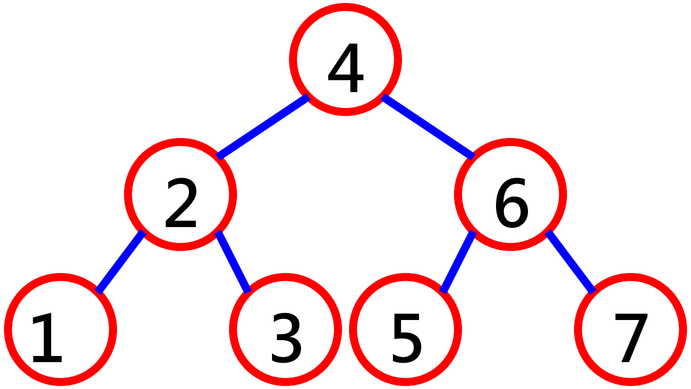
Worst case h=n

– Insert keys: 1, 2, 3, …

Best case h=log(n)

– Insert keys: 4, 2, 6, 1, 3, 5, 7





Summary

- Priority queue orders elements according to priority
  - Often queried for next highest priority element
    → more concerned with partial ordering
- PQ may be implemented efficiently using heap
  - Max/min heap can be implemented in turn using arrays
- O(log n) search/insertion/deletion of elements can be accomplished using binary search tree