Unit 2.2 Trees

Algorithms

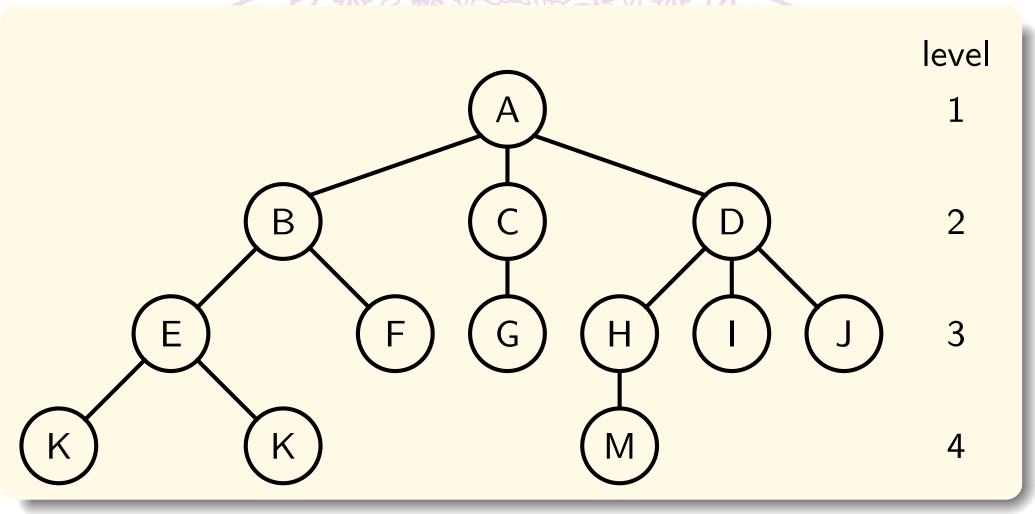
EE/NTHU

Mar. 11, 2019

Trees

Definition 2.2.1. Tree.

A *tree* is a finite set of one or more nodes such that there is a specially designated node called the *root* and the remaining nodes are partitioned into $n \geq 0$ disjoint sets T_1, ..., T_n, where each of these sets is a tree. The sets T_1, ..., T_n are called the *subtrees* of the root.



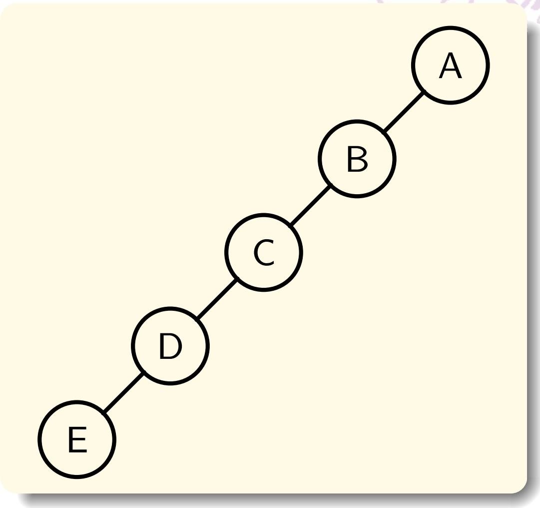
Trees, II

- The number of subtrees of a node is called its *degree*.
- Nodes that have degree 0 are called *leaf* or *terminal nodes*.
  - The other nodes are *nonterminals*.
- The roots of the subtree of a node X are the *children* of X.
  - The node X is the *parent* of its children.
- The *ancestors* of a node are all the nodes along the path from the root to that node.
- Children of the same parent are said to be *siblings*.
- The *degree* of a tree is the maximum degree of the nodes in the tree.
- The root is at *level* 1. If a node is at level p, then its children are at level $p+1$.
- The *height* or *depth* of a tree is the maximum level of any node in the tree.
- A *forest* is a set of $n > 0$ disjoint trees.

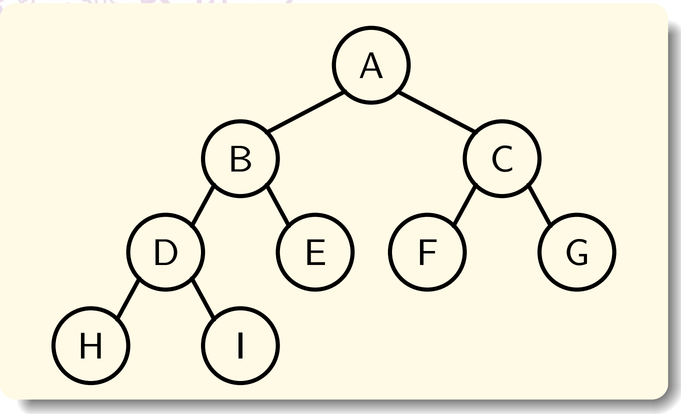
Binary Trees

Definition. 2.2.2. Binary Tree.

A *binary tree* is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the *left* and *right* subtrees.



A skewed binary tree.



A complete binary tree.

Dictionaries

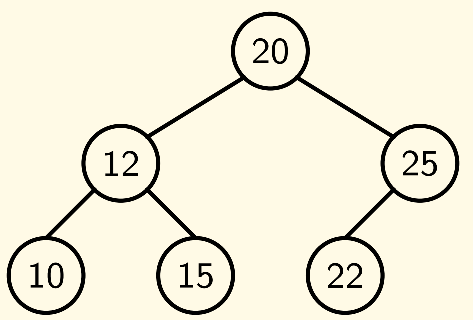
- An abstract data type that supports the operations *insert*, *delete* and *search* is called a *dictionary*.
- At high level dictionaries can be categorized as *comparison* methods and *direct access* methods.
  - Binary search tree is one of the comparison methods.
  - Hashing is an example of direct access method.

Binary Search Trees

Definition 2.2.3. Binary search tree.

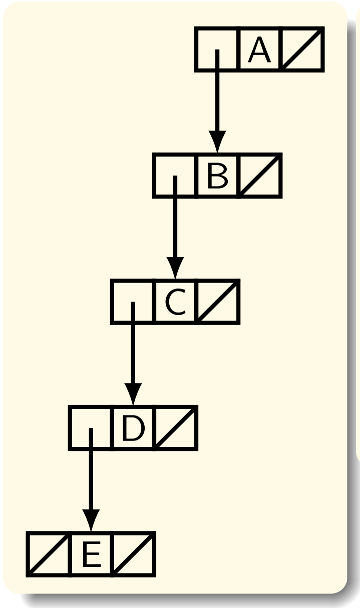
A *binary search tree* is a binary tree. It may be empty. If it is not empty, then it satisfies the following properties:

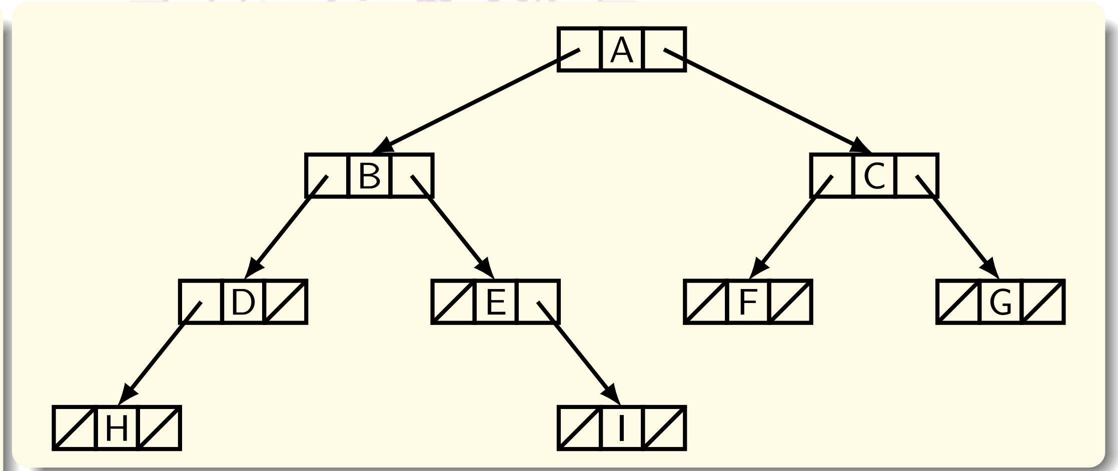
1. Every element has a key and no two elements have the same key (i.e., the keys are distinct).
2. The keys (if any) in the left subtree are smaller than the key in the root.
3. The keys (if any) in the right subtree are larger than the key in the root.
4. The left and right subtrees are also binary search trees.



Binary Search Trees – Data Structure

- Linked list can be used to store binary search trees.
- Each node has four items: *parent*, *lchild*, *rchild*, *key*.
  - An additional item, *leftsize*, is needed for search by rank algorithm.





Binary Search Trees – Min and Max

- Given a binary tree, the tree node with the minimum or maximum key can be found by the following algorithms.

Algorithm 2.2.4. Find the minimum in a Binary Search Tree

// Find the minimum in a binary tree T.
// **Input**: none; **Output**: min of T.
Algorithm BSTmin(T)
{
    $t := T$; **while** ($t \rightarrow lchild \neq$ NULL) $t := t \rightarrow lchild$;
    return t;
}

Algorithm 2.2.5. Find the maximum in a Binary Search Tree

// Find the maximum in a binary tree T.
// **Input**: none; **Output**: max of T.
Algorithm BSTmax(T)
{
    $t := T$; **while** ($t \rightarrow rchild \neq$ NULL) $t := t \rightarrow rchild$;
    return t;
}

Binary Search Trees – Search

- Search in a binary search tree can be easily done.
- This is a recursive version.

Algorithm 2.2.6. Recursive Search for a Binary Search Tree

```
// Recursive search key x in binary tree T.
// Input: binary tree T, key x; Output: the node with key x.
Algorithm BSTsearch_R(T, x)
{
    if (T = NULL or x = T→key) then return T ;
    if (x < T→key) then return BSTsearch_R(T→lchild, x) ;
    else return BSTsearch_R(T→rchild, x) ;
}
```

- For a binary tree of height h, the time complexity for all three algorithms above are $\mathcal{O}(h)$.

Binary Search Trees – Iterative Search

- Search in binary search tree can also be done iteratively.

Algorithm 2.2.7. Iterative Search for a Binary Search Tree

```
// Iterative search key x in binary tree T.
// Input: binary tree T, key x; Output: the node with key x.
Algorithm BSTsearch(T, x)
{
    t := T ;
    while (( t ≠ NULL ) and (x ≠ t→key)) do {
        if (x < t→key) then t := t→lchild;
        else t := t→rchild;
    }
    return t;
}
```

- The searching time is the same as the recursive version, $\mathcal{O}(h)$.

Binary Search Trees – Search by Rank

- If each node in the binary search tree has an additional item, *leftsize*, which is one plus the number of elements in the left subtree, then the following algorithm performs search by rank.

Algorithm 2.2.8. Search by Rank with Binary Search Tree

```
// Search the k-th element in binary tree T
// Input: binary tree T, rank k; Output: the k-th node in T.
Algorithm BSTsearchRank(T, k)
{
    t := T ;
    while ((t ≠ NULL ) and (k ≠ t→leftsize)) do {
        if (k < t→leftsize) then t := t→lchild;
        else {
            k := k − t→leftsize; t := t→rchild;
        }
    }
    return t;
}
```

- Time complexity is $\mathcal{O}(h)$.

Binary Search Trees – Successor

- The successor of a node in a binary tree can also be found in $\mathcal{O}(h)$ time.

Algorithm 2.2.9. Find the successor

```
// Find the successor of T in a binary tree.
// Input: node T; Output: successor of T.
Algorithm BSTsuccessor(T)
{
    if (T→rchild ≠ NULL) then
        return BSTmin(T→rchild);
    P := T→parent;
    while (P ≠ NULL and T = P→rchild) {
        T := P; P := P→parent;
    }
    return P;
}
```

- The predecessor can also be found similarly.

Binary Search Trees – Insertion

Algorithm 2.2.10. Binary Search Tree Insertion.

```
   // Insert a node with key x into the binary search tree T.
   // Input: tree T, key x; Output: updated tree T.
Algorithm BSTinsert(T, x)
{
    t := T; P := t→parent;
    while (t ≠ NULL ) { // Repeat until P is a leaf node.
        P := t;
        if (x < t→key) t := t→lchild; // Maintain BST property.
        else t := t→rchild;
    }
    q := new Node; // Create a new Node.
    q→lchild := NULL ; q→rchild := NULL ; q→key := x;
    if (P = NULL ) T := q; // The tree was empty.
    else if (x < P→key) P→lchild := q; // Insert.
    else P→rchild := q;
    return T;
}
```

- The time complexity is $\mathcal{O}(h)$.

Binary Search Trees – Deletion

- Delete a node need to consider the following cases:
  - Deletion of a leaf node is straightforward.
    - Remove the corresponding link from its parent.
  - Deletion of a nonleaf node that has only one child is also straightforward.
    - Replace the data of deleted node by its child's data
    - Then remove the child node.
  - Deletion of a nonleaf node that has two children can be done in the following way:
    - Replace the data of the deleted node by the largest element of its left subtree or the smallest element of its right subtree.
    - Then delete the replacing element from the subtree it is taken.
- Deletion of a binary search tree of height h can be done in $\mathcal{O}(h)$ time.

Binary Search Tree, Tree Height

- Given a binary tree with n nodes, then the maximum height is n.
- Thus the worst-case complexity of the above BST algorithms are $\mathcal{O}(n)$.
- However, we have the following theorem.

Theorem 2.2.11.

The expected height of a randomly built binary search tree on n distinct keys is $\mathcal{O}(\lg n)$.

- Proof please see textbook [Cormen], pp. 300-303.

- Thus, binary search tree is a good choice for dictionary applications.

Comparing to Other Trees

- More tree data structure have been proposed for dictionary applications.
- Worst-case $\mathcal{O}(\lg n)$ complexity can be achieved.

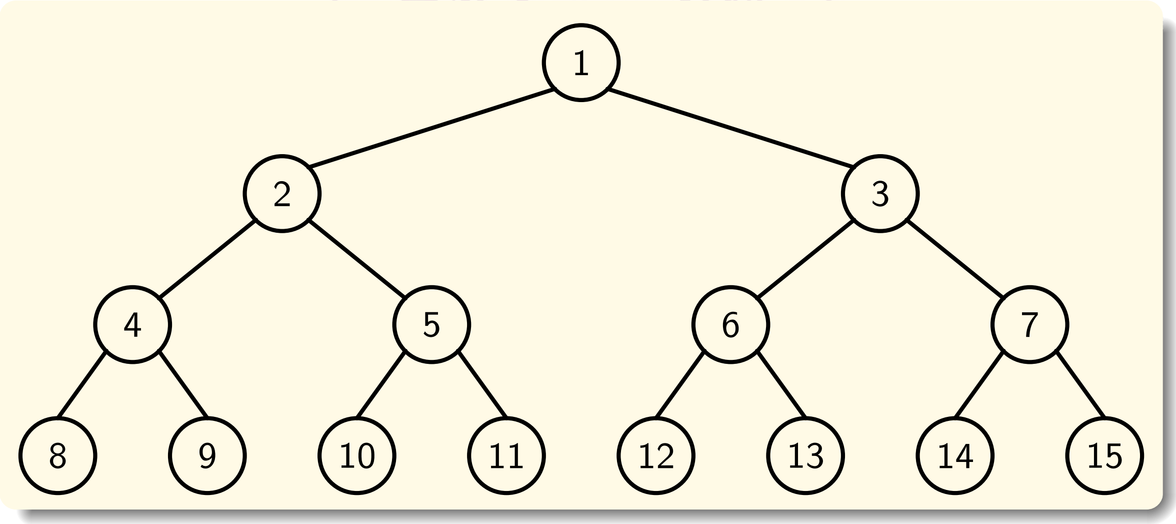
Data structure	Search	Insert	Delete
Binary search tree	$\mathcal{O}(n)$ (wc) $\mathcal{O}(\lg n)$ (av)	$\mathcal{O}(n)$ (wc) $\mathcal{O}(\lg n)$ (av)	$\mathcal{O}(n)$ (wc) $\mathcal{O}(\lg n)$ (av)
AVL tree	$\mathcal{O}(\lg n)$ (wc)	$\mathcal{O}(\lg n)$ (wc)	$\mathcal{O}(\lg n)$ (wc)
2-3 tree	$\mathcal{O}(\lg n)$ (wc)	$\mathcal{O}(\lg n)$ (wc)	$\mathcal{O}(\lg n)$ (wc)
Red-Black tree	$\mathcal{O}(\lg n)$ (wc)	$\mathcal{O}(\lg n)$ (wc)	$\mathcal{O}(\lg n)$ (wc)
B-tree	$\mathcal{O}(\lg n)$ (wc)	$\mathcal{O}(\lg n)$ (wc)	$\mathcal{O}(\lg n)$ (wc)
Splay tree	$\mathcal{O}(\lg n)$ (am)	$\mathcal{O}(\lg n)$ (am)	$\mathcal{O}(\lg n)$ (am)

(wc): worst case.
(av): average case.
(am): amortized cost.

Binary Trees – Maximum Nodes

Lemma 2.2.12.

The maximum number of nodes on level i of a binary tree is 2^{i-1}. Also, the maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k > 0$.



A full binary tree of depth 4.

Complete Binary Trees

- A binary tree of depth k has exactly $2^k - 1$ nodes is called a *full* binary tree of depth k.
- A full binary tree can be stored into a linear array with $2^k - 1$ elements. The root is stored in the first element, followed by its left child, and then the right child. All the nodes at the same level will be stored sequentially, from left to right.
- A binary tree with n nodes and depth k is complete if and only if its nodes correspond to the nodes that are numbered one to n in a full binary tree of depth k.
  - That is it can be stored in the first n elements of a linear array following the rules above.
  - In a complete binary tree, the leaf nodes occur on at most two adjacent levels.

Complete Binary Trees and Arrays

Lemma 2.2.13.

If a complete binary tree with n nodes is represented by a linear array, then for any node with index i, $1 \le i \le n$, we have:

1. $parent(i)$ is at $\lfloor i/2 \rfloor$, if $i \neq 1$. When $i = 1$, i is the root and has no parent.
2. $lchild(i)$ is at $2i$, if $2i \le n$. If $2i > n$, i has no left child.
3. $rchild(i)$ is at $2i+1$, if $2i+1 \le n$. If $2i+1 > n$, i has no right child.

- The linear array storage of complete binary tree is efficient with no waste.
  - But for general binary tree, there could be spaces wasted, especially for skewed trees.
  - Insertion and deletion of nodes are difficult to perform.

Priority Queues

- Any data structure that supports the operations of search min (or max), insert, and delete min (or max) is called a *priority queue*.

Definition 2.2.14. Heap

A *max (min) heap* is a complete binary tree with the property that the value at each node is at least as large as (as small as) the values at its children (if they exist). This property is called the *heap property*.

- By definition, the search time for max (or min) heap is $\mathcal{O}(1)$.
  - But, *insert* and *delete* function need to be carefully implemented.
- A max heap can be implemented using an array $A[1:n]$.
- The functions *insert* and *delete* are illustrated in the following.

Heap Insertion

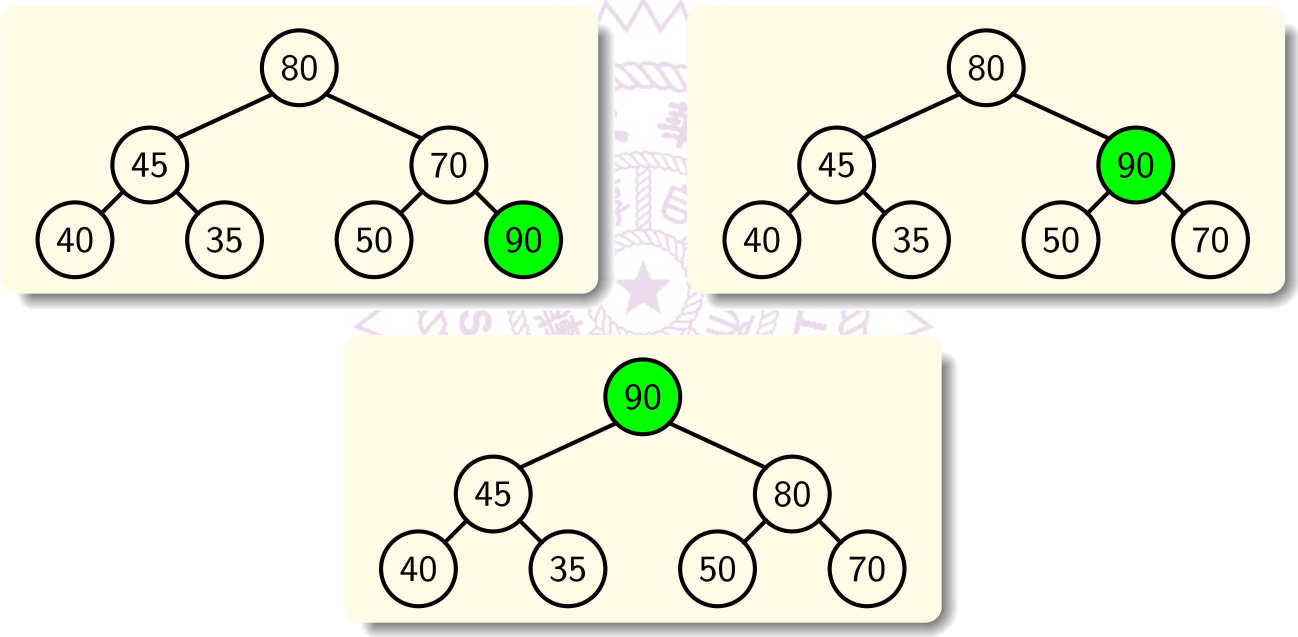
- The following algorithm insert an item to a max heap, which is represented by an array A.

Algorithm 2.2.15. Heap insertion.

```
// Insert the n-th element, item, to the max heap, A.
// Input: heat array A, int n, item; Output: update A.
Algorithm HeapInsert(A, n, item)
{
    i := n; A[n] := item; // initialization
    while ((i > 1) and (A[⌊i/2⌋] < item)) do { // maintain max heap property.
        A[i] := A[⌊i/2⌋]; i := ⌊i/2⌋; // parent should be larger.
    }
    A[i] := item;
}
```

- HeapInsert algorithm takes $\mathcal{O}(\lg n)$ time in worst case.
- Note that for a max heap, the root is always the largest element.
  - Also for all the subtrees.

Heap Insertion, Example



Heap Increase Key

- For an item in the max heap, some applications may need to increase the value (priority) of the item.
- The following algorithm perform such task and maintain the heap property.
- The time complexity if $\mathcal{O}(\lg n)$.

Algorithm 2.2.16. Heap Increase Key.

```
// Increase A[i] to key.
// Input: heap array A, int i, new key; Output: updated A.
Algorithm HeapIncKey(A, i, key)
{
    if (A[i] > key) error ( "new key is smaller" ) ;
    A[i] := key ; // increase key.
    while (i > 1 and A[⌊i/2⌋] < A[i]) do { // maintain max heap property.
        t := A[i] ; A[i] := A[⌊i/2⌋] ; A[⌊i/2⌋] := t; // swapping keys.
        i := ⌊i/2⌋ ;
    }
}
```


Heap Remove Max

- The following algorithm remove the maximum from the max heap and then calls Heapify to maintain the max heap property.
- It can be shown that the complexity is also $\mathcal{O}(\lg n)$.

Algorithm 2.2.17. Heap Remove Max.

```
// Remove and return the maximum of the heap array A[1 : n].
// Input: max heap array A, int n; Output: max value and updated A.
Algorithm HeapRmMax(A, n)
{
    if (n = 0) then error (" heap is empty! ") ;
    x := A[1] ; A[1] := A[n] ;
    Heapify(A, 1, n − 1) ;
    return x ;
}
```

Heapify – Maintain Heap Property

Algorithm 2.2.18. Maintain heap property

```
// To enforce max heap property for n-element heap A with root i.
// Input: size n max heap array A, root i; Output: updated A.
Algorithm Heapify(A, i, n)
{
    j := 2×i; item := A[i]; done := false ; // A[j] is the lchild.
    while ((j ≤ n) and ( not done )) do { // A[j+1] is the rchild.
        if ((j < n) and (A[j] < A[j+1])) then
            j := j+1 ; // A[j] is the larger child.
        if (item > A[j]) then // If larger than children, done.
            done := true ;
        else { A[⌊j/2⌋] := A[j] ; j := 2×j; } // Otherwise, continue.
    }
    A[⌊j/2⌋] := item;
}
```

- The algorithm compares the value of the root with its children.
    - If not larger, moves the larger value to the root and continue downwards.
- The time complexity is $\mathcal{O}(\lg n)$.

Heap Sort

- The HeapRmMax(A, n) algorithm removes the largest element from the array A, and then the algorithm Heapify$(A, 1, n-1)$ adjusts the array $A[1 : n-1]$ such that it satisfies the max heap property.
- Removing the largest element takes $\mathcal{O}(1)$ time, and maintaining the max heap property takes $\mathcal{O}(\lg n)$ time.
- Thus, one can use these algorithms to perform sort function.
- In order to do that, the array needs to satisfy max heap property first.
- The Heapify algorithm can also be use for this job.
    - Starting from the deepest internal nodes down to the root, perform Heapify on these internal nodes.
    - Leave nodes have no *lchild* nor *rchild*, and thus no need to perform Heapify on them.
    - Around $n/2$ nodes to Heapify and each takes $\mathcal{O}(\lg n)$ time.
    - Total complexity is $\mathcal{O}(n \lg n)$.
- After that one can remove the maximum element and then perform Heapify to maintain the max heap property.
    - This process repeats until the entire A array is sorted.
    - Heapify is called n times and each iteration take $\mathcal{O}(\lg n)$ time.
    - Total time complexity is $\mathcal{O}(n \lg n)$.

Heap Sort

Algorithm 2.2.19. Heap sort.

```
// Sort A[1 : n] into nondecreasing order.
// Input: Array A with n elements; Output: A sorted in nondecreasing order.
Algorithm HeapSort(A, n)
{
    for i := ⌊n/2⌋ to 1 step −1 do // Initialize A[1 : n] to be a max heap.
        Heapify(A, i, n);
    for i := n to 2 step −1 do { // Repeat n − 1 times
        t := A[i]; A[i] := A[1]; A[1] := t; // Move maximum to the end.
        Heapify(A, 1, i − 1); // Then make A[1 : i − 1] a max heap.
    }
}
```

- The time complexity of $\mathcal{O}(n \lg n)$
  – The best possible for comparison based sorting algorithms.

Date Structures for Priority Queue

- Priority queues have many applications.
- Various data structures that support priority queue.

Data Structure	Insert	Remove max/min
Min heap	$\mathcal{O}(\lg n)$ (wc)	$\mathcal{O}(\lg n)$ (wc)
Min-max heap	$\mathcal{O}(\lg n)$ (wc)	$\mathcal{O}(\lg n)$ (wc)
Deap	$\mathcal{O}(\lg n)$ (wc)	$\mathcal{O}(\lg n)$ (wc)
Leftist tree	$\mathcal{O}(\lg n)$ (wc)	$\mathcal{O}(\lg n)$ (wc)
Binomial heap	$\mathcal{O}(\lg n)$ (wc) $\mathcal{O}(1)$ (am)	$\mathcal{O}(\lg n)$ (wc) $\mathcal{O}(\lg n)$ (am)
Fibonacci heap	$\mathcal{O}(\lg n)$ (wc) $\mathcal{O}(1)$ (am)	$\mathcal{O}(\lg n)$ (wc) $\mathcal{O}(\lg n)$ (am)
2-3 tree	$\mathcal{O}(\lg n)$ (wc)	$\mathcal{O}(\lg n)$ (wc)
Red-Black tree	$\mathcal{O}(\lg n)$ (wc)	$\mathcal{O}(\lg n)$ (wc)

Summary

- Trees
- Binary search tree
  - Recursive and iterative searches
  - Insert and delete
  - Application: dictionary
- Heap
  - Max and min heap
  - Insert and delete
  - Application: priority queue
  - Heap sort