

Efficient Algorithms

Ekkapot Charoenwanit

Software Systems Engineering

TGGS

KMUTNB

Lecture 3: Data Structures (Part I)

Stack, Queue, Heap, Priority Queue

Stack

A stack S consists of elements $S[1 \dots S.top]$

- $S.top$ points to the **most recently** inserted element
- $S[1]$ points to the element at the bottom of the stack, i.e., the **oldest** element
- A stack exhibits **LIFO** behavior.
 - A new item is added to the top of the stack
 - Only the most recently item $S[S.top]$ can be deleted or popped off the stack

Stack Operations

```
1: procedure STACKEMPTY( $S$ )
2:   if  $S.top == 0$  then
3:     return True
4:   else
5:     return False
```

```
1: procedure PUSH( $S, x$ )
2:    $S.top = S.top + 1$ 
3:    $S[S.top] = x$ 
```

```
1: procedure POP( $S$ )
2:   if STACK-EMPTY( $S$ ) then
3:     error "UNDERFLOW"
4:   else
5:      $S.top = S.top - 1$ 
6:     return  $S[S.top + 1]$ 
```

The Time Complexity of Stack Operations

- Efficient operations in $O(1)$ time:
 - ***PUSH, POP***
 - ***TOP, EMPTY, SIZE***
- Applications:
 - Tree Traversal and Backtracking
 - Evaluating arithmetic expressions
 - Parsing Grammar (CFG)
 - Implementation of Function Call Mechanism
- Implementation:
 - Array
 - Linked List

Queue

A queue Q exhibits **FIFO** behavior.

- $Q.head$ points to the start of the queue
- $Q.tail$ points to the end of the queue

You may think of a queue this way:

- The element pointed to by $Q.head$ has the highest priority
- The element pointed to by $Q.tail$ has the lowest priority

Initially, $Q.head = Q.tail = 1$, which means the queue is empty.

Queue Operations

```
1: procedure ENQUEUE( $Q, x$ )
2:   if  $Q.tail == Q.head$  then
3:     error "OVERFLOW"
4:    $Q[Q.tail] = x$ 
5:   if  $Q.tail == Q.length$  then
6:      $Q.head = 1$ 
7:   else
8:      $Q.head = Q.head + 1$ 
```

```
1: procedure DEQUEUE( $Q$ )
2:   if  $Q.head == Q.tail$  then
3:     error "UNDERFLOW"
4:    $x = Q[Q.head]$ 
5:   if  $Q.head == Q.length$  then
6:      $Q.head = 1$ 
7:   else
8:      $Q.head = Q.head + 1$ 
9:   return  $x$ 
```

The Enqueue and Dequeue operations take $O(1)$ constant time.

Binary Heap

A **binary heap** (or simply **heap**) is an *array* which represents an *ordered binary tree*:

- all levels of a binary tree are filled *except* possibly for the last level, which is filled from left to right
 - This property is referred to as being a *Nearly Complete Binary Tree*.
- values stored at each node obey *the MIN/MAX Heap Property*, depending on which kind of heap the heap is.

Binary Heap

In a binary heap,

$A[1]$ is the root of the binary tree

Given the index i of a node, we can compute the indices of:

- the parent
- the left child
- the right child

Parent-Child Relationship

```
1: procedure PARENT( $i$ )  
2:   return  $\lfloor \frac{i}{2} \rfloor$ 
```

```
1: procedure LEFT( $i$ )  
2:   return  $2i$ 
```

```
1: procedure RIGHT( $i$ )  
2:   return  $2i + 1$ 
```

******Assume array indexing starts at 1.***

The Max Heap Property

For every node i *other than the root*, the value of a node is at most the value of its parent

$$A[\text{Parent}(i)] \geq A[i]$$

Therefore, the *maximum value* is stored at the root $A[1]$.

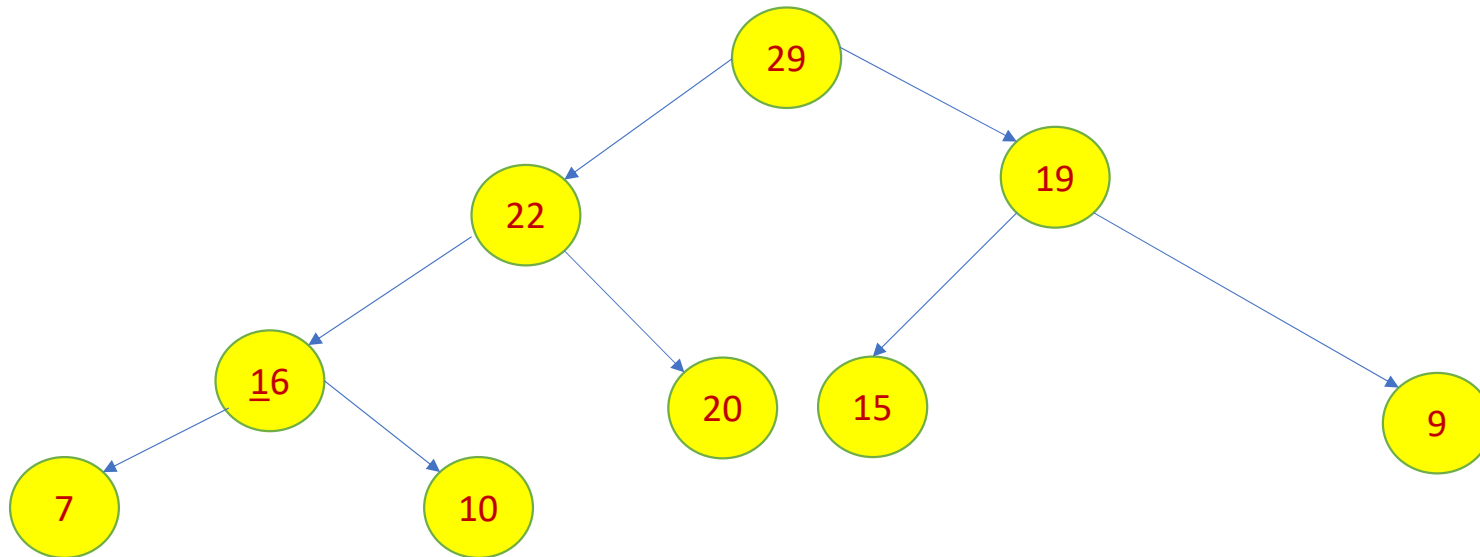
For a *min heap*, the *min heap property* is the opposite:

$$A[\text{Parent}(i)] \leq A[i]$$

******From now on, we will talk about only the max heap.***

The Max Heap Property

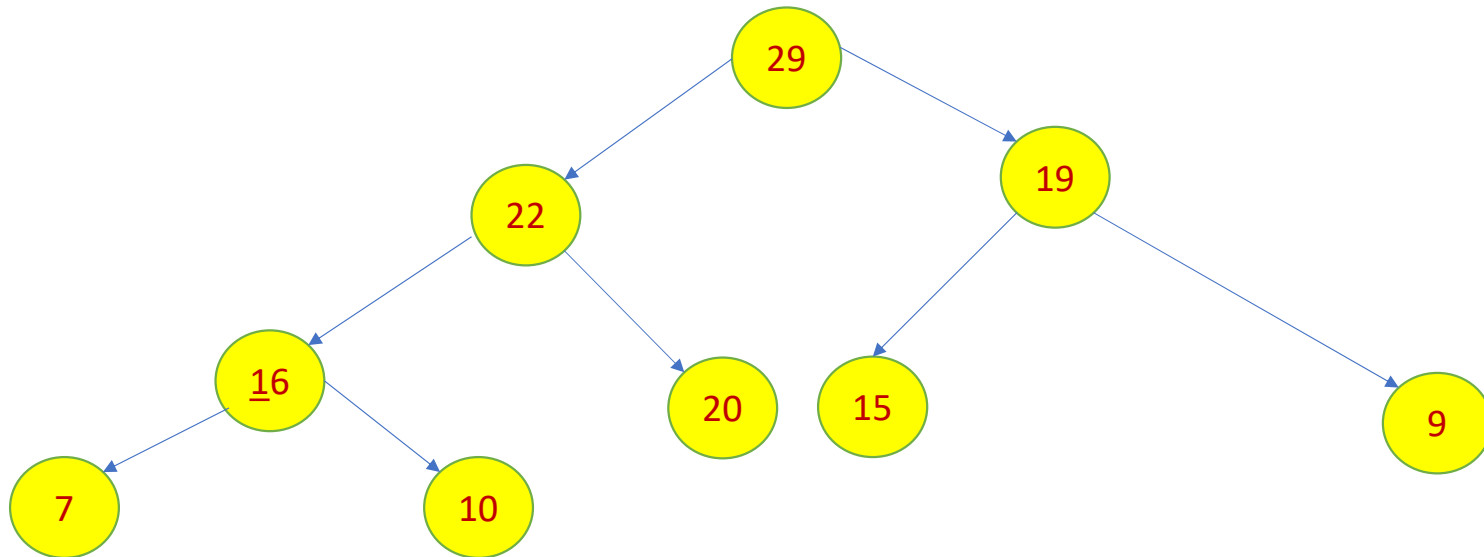
Is this array a max heap ***[29,22,19,16,20,15,9,7,10]***?



Check if the max heap property is satisfied at every node.

The Max Heap Property

Is this array a max heap ***[29,22,19,16,20,15,9,7,10]***? ***YES !!!***



Definition of the Height of a Tree

Definition I:

The height of a node is the number of edges on the longest simple path from that node to a leaf.

Definition II:

The height of a tree is the height of the root.

Definition of the Height of a Tree

Since a **binary heap** can be viewed as a **nearly complete** binary tree, the **height** h of the heap is

$$h = \Theta(\log n)$$

,where n is the number of elements in the heap.

We will prove that some **basic operations** on any binary heap run in time proportional to **the height of the heap** h .

Therefore, the time complexity $T_{Op}(n)$ of these basic operations is $O(\log n)$.

How to prove $h = \Theta(\log n)$

First, we must show that the minimum and the maximum number of elements n in a heap of height h are

2^h and $2^{h+1} - 1$, respectively.

In other words,

$$2^h \leq n \leq 2^{h+1} - 1$$

Then, we must show that an n -element heap has height $\lfloor \log_2 n \rfloor$, which means $h = \Theta(\log n)$.

Max-Heapify

Suppose there is a binary heap rooted at node i .

Before *MaxHeapify* is called on the tree rooted at node i , the following precondition needs to be met.

Precondition: If there will be a violation on the max heap property within this tree, it can only happen at node i . In other words, both trees rooted at $Left(i)$ and $Right(i)$ obey the max heap property.

We can fix up this violation using *MaxHeapify* shown on the right.

```
1: procedure MAX-HEAPIFY( $A, i$ )
2:    $l = Left(i)$ 
3:    $r = Right(i)$ 
4:   if  $l \leq A.HeapSize \wedge A[l] > A[i]$  then
5:      $largest = l$ 
6:   else
7:      $largest = i$ 
8:   if  $r \leq A.HeapSize \wedge A[r] > A[largest]$  then
9:      $largest = r$ 
10:  if  $largest \neq i$  then
11:     $A[i] \iff A[largest]$ 
12:    MAX-HEAPIFY( $A, largest$ )
```

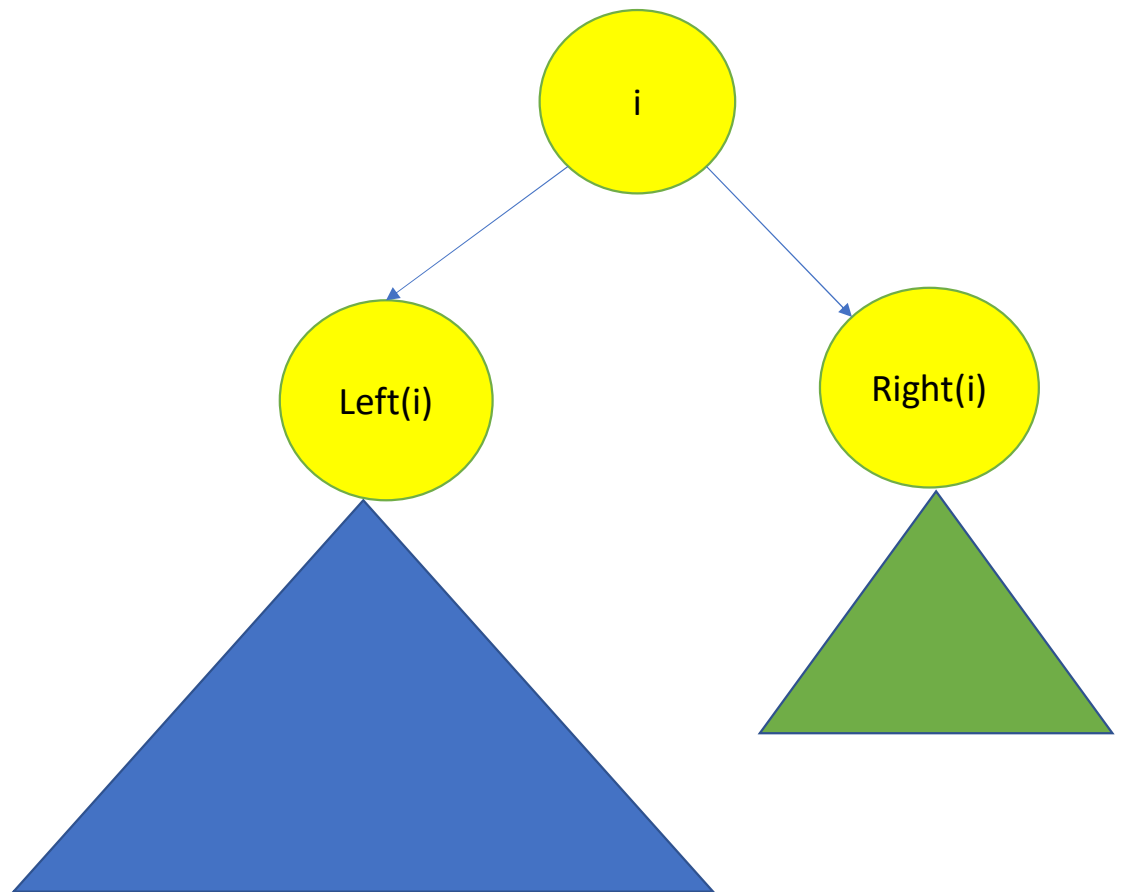
Max-Heapify

Suppose there is a binary heap rooted at node i .

Before *MaxHeapify* is called on the tree rooted at node i , the following precondition needs to be met.

Precondition: If there will be a violation on the max heap property within this tree, it can only happen at node i . In other words, both trees rooted at $Left(i)$ and $Right(i)$ obey the max heap property.

We can fix up this violation using *MaxHeapify*.



Time Complexity of Max-Heapify

The **time complexity** $T(n)$ of *MaxHeapify* on a tree of n elements rooted at a given node i is the height of the tree in **the worst case**.

$$T(n) = O(\log n)$$

Another way to prove this is to derive the running time $T(n)$ as a recurrence relation in terms of the running time of the recursive call on a subtree rooted at one of the children of node i plus the work done at each level of a recursive call.

Build-Max-Heap

We can use *MaxHeapify* in a **bottom-up manner** to convert an array $A[1 \dots n]$ into a max heap as follows.

```
1: procedure BUILD-MAX-HEAPIFY( $A$ )
2:    $A.HeapSize = A.Length$ 
3:   for  $i = A.Length \rightarrow 1$  do
4:     MAX-HEAPIFY( $A, i$ )
```

But, can we do better?

Build-Max-Heap

We can use *MaxHeapify* in a **bottom-up manner** to convert an array $A[1 \dots n]$ into a max heap as follows.

```
1: procedure BUILD-MAX-HEAPIFY( $A$ )
2:    $A.HeapSize = A.Length$ 
3:   for  $i = \left\lfloor \frac{A.Length}{2} \right\rfloor \rightarrow 1$  do
4:     MAX-HEAPIFY( $A, i$ )
```

Key Observation: the elements in the subarray $A[\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n]$ are all leaves of the tree.

Build-Max-Heap

Claim: The elements in the subarray $A[\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n]$ are all leaves of the tree.

Proof: We will prove by contradiction.

Assume the node with index $\lfloor n/2 \rfloor + 1$ is not a leaf.

Therefore, it must have a left child whose index is

$$2(\lfloor n/2 \rfloor + 1) = 2\lfloor n/2 \rfloor + 2.$$

Build-Max-Heap

Proof:

We know that

$$\lfloor n/2 \rfloor > n/2 - 1$$

$$2\lfloor n/2 \rfloor > n - 2$$

$$2\lfloor n/2 \rfloor + 2 > n$$

We have just found that the index of the left child is bigger than n , which is a contradiction to the fact that there are only n elements in the heap.

Therefore, the node with index $\lfloor n/2 \rfloor + 1$ is a leaf.

Therefore, the nodes with larger indices must also be leaves. ■

Build-Max-Heap

Although we have shown that $A[\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n]$ are ***all leaves***,

How do we confirm that the node with index $\lfloor n/2 \rfloor$ is ***not a leaf***?

******Left as homework (PS 3.2.3)***

Time Complexity of Build-Max-Heap

- Each call to *MaxHeapify* costs $O(\log n)$.
- *BuldMaxHeap* makes $\lceil n/2 \rceil = O(\log n)$ such calls.
- The time complexity is $O(n \log n)$.
- This upper bound is correct, but it is not the tightest asymptotic claim we can make.
- We can make a tighter claim based on the following claim:

Claim:

A binary heap with n elements can have at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of any height h .

Mathematical Induction on the Height

Claim:

A binary heap with n elements can have at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of any height h .

Proof: We will prove by induction on the height h .

Base Case: $h = 0$

All the nodes at height $h = 0$ are the leaf nodes.

There are $\left\lceil \frac{n}{2} \right\rceil$ leaves (***You will prove this in PS 3.2.4***), which is $\leq \left\lceil \frac{n}{2^{0+1}} \right\rceil = \left\lceil \frac{n}{2} \right\rceil$.

We are done with the base case.

Time Complexity of Build-Max-Heap

Induction Hypothesis: Assume true for any $0 \leq h \leq k - 1$.

Inductive Step: Show true for $h = k$.

Let n_k be the number of nodes at height k in a tree T with n nodes.

Construct another tree T' by removing the leaves of T .

Therefore, T' has $n' = n - \left\lceil \frac{n}{2} \right\rceil = \left\lfloor \frac{n}{2} \right\rfloor$ nodes.

The nodes at height k in T will become the nodes at height $k - 1$ in T' .

Let n'_{k-1} be the number of nodes at height $k - 1$ in T' .

Therefore, $n_k = n'_{k-1} \leq \left\lfloor \frac{n'}{2^{(k-1)+1}} \right\rfloor$ (I.H. at $k - 1$ for n')

Time Complexity of Build-Max-Heap

Inductive Step:

$$\begin{aligned}n_k = n'_{k-1} &\leq \left\lceil \frac{n'}{2^{(k-1)+1}} \right\rceil \\&= \left\lceil \frac{n'}{2^k} \right\rceil \\&= \left\lceil \frac{\left\lfloor \frac{n}{2} \right\rfloor}{2^k} \right\rceil \\&\leq \left\lceil \frac{\frac{n}{2}}{2^k} \right\rceil \\&= \left\lceil \frac{n}{2^{k+1}} \right\rceil\end{aligned}\quad \left(\left\lfloor \frac{n}{2} \right\rfloor \leq \frac{n}{2} \forall n \in \mathbb{R} \right)$$

We have just shown true for $h = k$ that $n^k \leq \left\lceil \frac{n}{2^{k+1}} \right\rceil$. ■

Time Complexity of Build-Max-Heap

- A binary heap with n elements can have at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of any height h .
- Each *MaxHeapify* costs $O(h)$ for a node of height h .
- We know that $0 \leq h \leq \lfloor \log_2 n \rfloor$.

The time complexity of *MaxBuildHeap* is bound from above by

$$\sum_{i=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(\frac{n}{2} \sum_{i=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{i=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h}\right)$$

Time Complexity of Build-Max-Heap

Observation:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2.$$

Therefore, $\sum_{i=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(2n) = O(n)$ ■

Thus, we can build a max heap from an unordered array in **linear time** $O(n)$.

Heap Sort

- First, we build a max heap from an unordered array A (Line 2).
- The **maximum element** is initially stored at the **root** $A[1]$.
- We put the maximum into the correct position by swapping $A[1]$ with $A[i]$, where i points to the last element of the heap in the current iteration (Line 4).
- We can now reduce the size of our max heap by decrementing the heap size by one (Line 5).
- We fix up the potential violation at the root $A[1]$ with *MaxHeapify* (Line 6).
- If we perform such a swap $n - 1$ times, the array A will be sorted in increasing order.

```
1: procedure HEAP-SORT( $A$ )
2:   BUILD-MAX-HEAP( $A$ )
3:   for  $i = A.Length \rightarrow 2$  do
4:      $A[1] \iff A[i]$ 
5:      $A.HeapSize = A.HeapSize - 1$ 
6:     MAX-HEAPIFY( $A, 1$ )
```

The time complexity of Heap Sort

Analysis:

BuildMaxHeap costs $T_1(n) = O(n)$ time.

Each call to *MaxHeapify* costs $O(\log k)$ time for $k = n, n - 1, \dots, 2$.

- The total cost of *MaxHeapify* $T_2(n)$ is

$$T_2(n) = \sum_{k=2}^n O(\log k) = O\left(\sum_{k=1}^n \log k\right) = O(\log n!)$$

Key Observation: $\log n! \leq \log n^n = n \log n \quad \forall n \geq 1$

$$T_2(n) = O(\log n^n) = O(n \log n)$$

$$T(n) = T_1(n) + T_2(n) = O(n) + O(n \log n) = O(n \log n)$$

Priority Queue

One popular use of binary heaps is to implement a **priority queue**.

A **priority queue** is a data structure for maintaining a set S of elements, each of which is assigned a value called **key**.

Priority queues come into **two variants**: **max** and **min** priority queue, depending on the type of the underlying binary heap.

*****In this lecture, we will talk about the max priority queue.**

Basic Operations

- *Maximum*(S) returns the element of S with the largest key.
- *ExtractMax*(S) removes and returns the element of S with the largest key.
- *IncreaseKey*(S, x, k) increases the value of element x 's key to the new value k , which is assumed to be at least as large as x 's current key value.
- *Insert*(S, x) inserts the element x into the set S .

Maximum

An element with the *largest key value* is stored at the *root* $A[1]$.
The operation has a constant time complexity $O(1)$.

```
1: procedure MAXIMUM( $A$ )  
2:   return  $A[1]$ 
```

Extract-Max

The time complexity of *ExtractMax* is $O(\log n)$ since it performs only constant time operations before it performs *MaxHeapify* on the root $A[1]$, which costs $O(\log n)$.

- 1) Save the value of the root element **(Line 4)**
- 2) Swap the element at the root with the last leaf element. **(Line 5)**
*****At this point, the max heap property might have been violated.**
- 3) Decrease the size of the heap by one, which effectively discards the extracted max element **(Line 6)**
- 4) Fix up the heap by calling *MaxHeapify* on the root **(Line 7)**
- 5) Return the extracted element **(Line 8)**

```
1: procedure EXTRACT-MAX(A)
2:   if  $A.heapSize < 1$  then
3:     error "UNDERFLOW"
4:    $max = A[1]$ 
5:    $A[1] = A[A.heapSize]$ 
6:    $A.heapSize = A.heapSize - 1$ 
7:   MAX-HEAPIFY(A, 1)
8:   return  $max$ 
```

Increase-Key

An index i is to point to the element $A[i]$ whose key value we want to increase.

Key Assumption: The new key key value must be at least as large as the current key value of $A[i]$. Otherwise, it is an error. **(Lines 2 & 3)**

If the new key value is ok,

1) Assign $A[i]$ the new key value key **(Line 4)**

*****At this point, the max heap property might have been violated.**

2) Traverse a simple path from $A[i]$ towards the root to find a proper place for the new key value **(Lines 5&6&7)**

```
1: procedure INCREASE-KEY( $A, i, key$ )
2:   if  $A[i] > key$  then
3:     error "INVALID - KEY - VALUE"
4:    $A[i] = key$ 
5:   while  $i > 1 \wedge A[Parent(i)] < A[i]$  do
6:      $A[i] \iff A[Parent(i)]$ 
7:      $i = Parent(i)$ 
```

The time complexity is determined by the length of the path traversed up the tree, which is $O(\log n)$.

Max-Heap-Insert

MaxHeapInsert takes as input the key of a new element *key* to be inserted into the max heap *A*.

1. Expand the max heap by incrementing the size by one **(Line 2)**
2. Add a new element with key value $-\infty$ **(Line 3)**
3. Call *IncreaseKey* to set the key value of the newly inserted element to *key* and make sure the max heap property is maintained **(Line 4)**

```
1: procedure MAX-HEAP-INSERT(A, key)
2:   A.heapSize = A.heapSize + 1
3:   A[A.heapSize] =  $-\infty$ 
4:   INCREASE-KEY(A, A.heapSize, key)
```

The time complexity of *MaxHeapInsert* is $O(\log n)$ since it performs constant time work before it calls *Increasekey*, which runs in $O(\log n)$ time.

Summary

We have covered the following data structures today:

- Stack
- Queue
- Heap
- Priority Queue

We have also illustrated an application of mathematical induction on discrete structures.

We will cover **Part II** of Data Structures in the next lecture.