Heaps

- Last time
  - Solving recurrences
- Today
  - Heaps

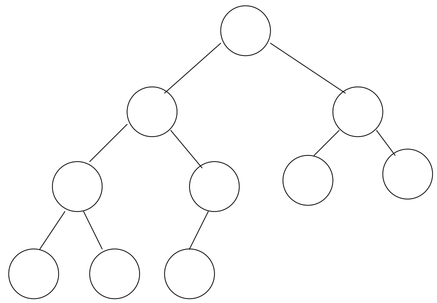
Reviews: Tree

- Tree
  - Rooted tree
    - parent, child, sibling, ancestor
- Binary tree
  - Left child, right child
- Some concepts
  - Height
  - Depth
  - Level
    - Level(n)= Height(root)-depth(n)

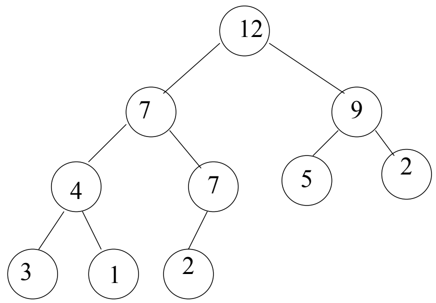
Heap Definition

- A heap is
  - An *essentially complete* *binary* tree which satisfies heap property.
- Binary tree
- Essentially complete binary tree
- Heap property
  - max-heap
    - The value (key) of each node in the heap is greater than or equal to the values (keys) of its children, if any.
  - min-heap
    - The value (key) of each node in the heap is less than or equal to the values (keys) of its children, if any.

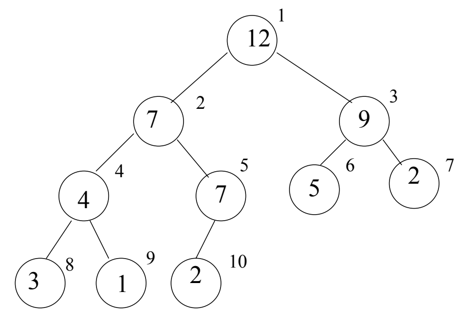
An essentially complete binary tree



A max-heap



A heap can be represented as an array



1	2	3	4	5	6	7	8	9	10
12	7	9	4	7	5	2	3	1	2

Some important properties of heaps

- Given a node *A[i]*
  - It's parent is *A[i/2]*, if *i>1*.
  - It's left child is *A[2*i]*, if *2*i<=n*.
  - It's right child is *A[2*i+1]*, if *2*i+1<=n*.
- The height of a heap containing n nodes is $\lfloor \lg n \rfloor$
- There are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes with height h

Methods of class MaxHeap

```
Class MaxHeap {
  int A[];
  int n;

  public void heapify(int i);
  public void increaseKey(int i, int key);
  public int maximum();
  public int extractMax();
  public void insert(int key);
  public void buildHeap();
  public void heapSort();
}
```

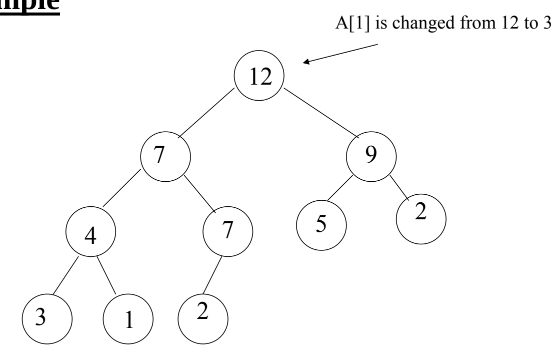
Heapify

- Assume that the left and right subtrees of A[i] are already max-heaps
- A[i] may be less than its children a violation
- Goal: Make the subtree rooted at index i a max-heap
- Application
  - Call heapify(i) when the value of A[i] is decreased

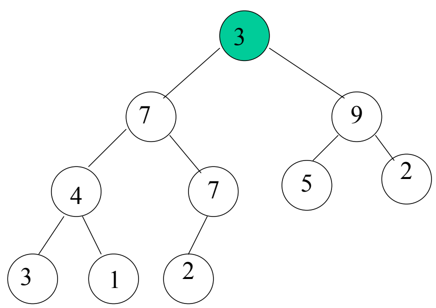
```
heapify(int i)   // also called sift-down
{
  int largest = i;
  int parent, lchild, rchild;

  do {
     parent = largest;
     lchild = 2*parent;
     if  (lchild <= n && A[lchild]>A[largest])
        largest = lchild;
     rchild = lchild++;
     if  (rchild <= n && A[rchild]>A[largest])
          largest = rchild;
      swap(A[parent], A[largest]);
  } while (parent != largest);
}
```

Cost?

Example

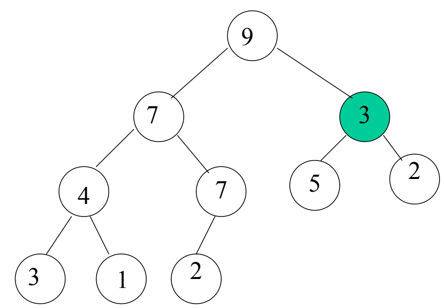


Example

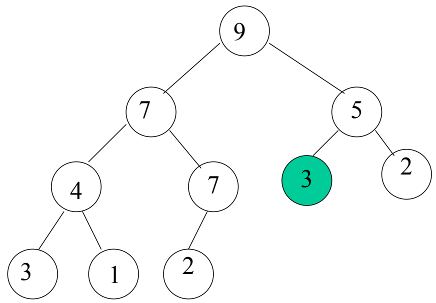


Call heapify(1)

Example



Example

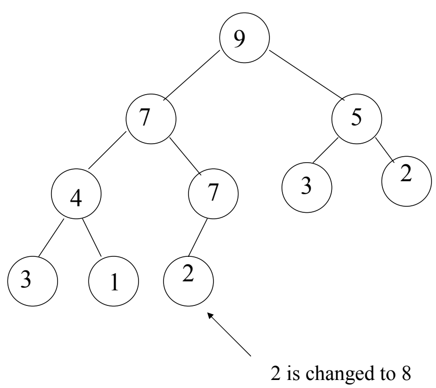


increaseKey

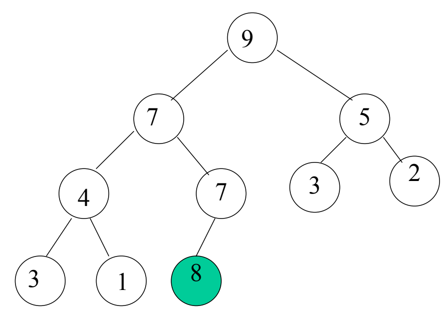
```
public void inreaseKey(int i, int key)
{
   int cur = i;
   int parent;

   A[cur] = key;
   do {
      parent = cur/2;
      if (parent > 0 && A[cur]>A[parent]){
         swap(A[cur], A[parent]);
         cur = parent;
      } else
         break;
   } while(1);
}
```

Cost?

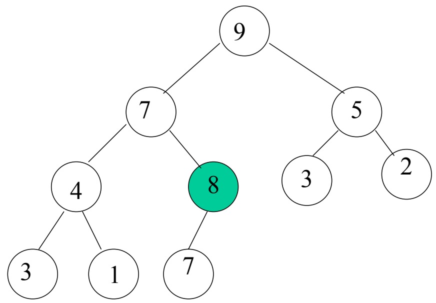
Example: increaseKey



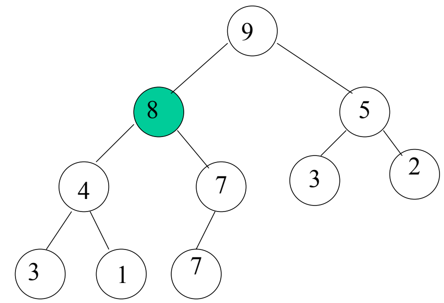
Example: increaseKey



Example: increaseKey



Example: increaseKey



maximum

```
int maximum()
{
   return A[1];
}
```

extractMax

```
int extractMax ()
{
   int max = A[1];

   A[1] = A[n];
   n = n-1;   // change heap size
   heapify(1);
   return max;
}
```

insert

```
insert(int key)
{
   n = n+1;   // change heap size
   A[n] = -∞;
   increaseKey(n, key);
}
```

Efficiency

```
Class MaxHeap {
  int T[];
  int n;

  public void heapify(int i); // O(lg n)
  public void increseKey (int i);          // O(lg n)
  public int maximum();                    // Θ(1)
  public int extractMax();               // O(lg n)
  public void insert(int key);             // O(lg n)
  public void buildHeap();               // O(n)
  public void heapSort();                // O(n lgn)
}
```

slowBuildMaxHeap

```
void slowBuildHeap()
{
   for (i=2; i<=n; i++)
     insert(A[i]);
}
```

Cost: homework.

buildHeap

```
void buildHeap()
{
   for (i = n/2; i >= 1; i--) {
      heapify(i);
   }
}
```

- What's the idea here?
- Proof

Analysis

```
void buildHeap()
{
   for (i=n/2; i>=1; i--) {
      heapify(i);
   }
}
```

```
heapify(int i)   // also called sift-down
{
  int largest = i;
  int parent, lchild, rchild;

  do {
     parent = largest;
     lchild = 2*parent;
     if  (lchild <= n && A[lchild]>A[largest])
        largest = lchild;
     rchild = lchild++;
     if  (rchild <= n && A[rchild]>A[largest])
        largest = rchild;
      swap(A[parent], A[largest]);
  } while (parent != largest);
}
```

loop iterations <= level of node i + 1

Analysis cont.

Total loop iterations:

$$t(n) \le 2 * 2^{k-1} + 3 * 2^{k-2} + \ldots + (k+1)2^0$$
$$\le 3 * n$$

An alternative analaysis

- The cost of heapify(i) for a node at height h is O(h)
- The total cost is bounded by

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

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \sum_{h=0}^{\infty} h(\frac{1}{2})^h = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$$

heapSort

```
void heapSort()
{
  buildHeap();
  tmp=n;
  for (i=n; i>=2; i--) {
     exchange(A[1],A[i]);
     n = i-1; // current heap size
     heapify(1);
  }
  n=tmp;
}
```

Cost: O(nlgn)

Application: Priority Queues

- insert(S, x)
  - Insert element x into S
- maximum(S)
  - Return the element with the largest key
- extractMax(L)
  - Remove and return the largest element
- increaseKey(S, x, k)
  - Increase the value of element x's key to the new value k

Implementation

- Use a MaxHeap H to implement the priority queue
  - insert(S, x)
    - H.insert(x)
  - maximum(S)
    - H.maximum()
  - extractMax(L)
    - H.extractMax()
  - increaseKey(S, x, k)
    - H.increaseKey(i, k); // use an index to represent an element