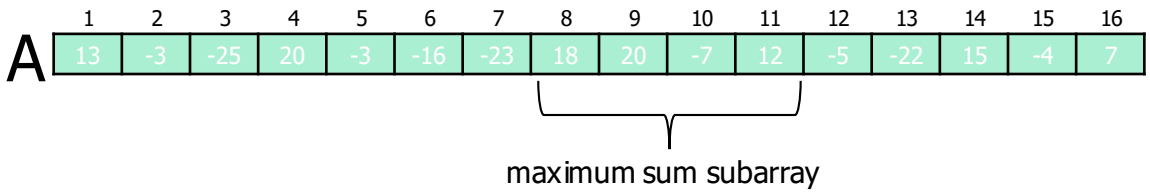
Maximum Subarray Problem

Maximum Subarray Problem

- Given an array A with integers,
- Find the contiguous subarray of A whose values have the maximum sum ($A[i]+A[i+1]+...+A[j]$)
- The maximum sum is zero if all the integers are negative.
- Number of possible ranges [i, j] for n numbers in the array A?



Maximum Subarray Problem

- Number of possible ranges [i, j] for n numbers in the array A?

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

Maximum Subarray Problem

- Number of possible ranges [i, j] for n numbers in the array A?
  - i = 1, j = 1, 2, …, n
  - i = 2, j = 2, 3, …, n
  - …
  - …
  - i = n-1, j = n-1, n
  - i = n, j = n

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

Maximum Subarray Problem

- Number of possible ranges [i, j] for n numbers in the array A?
  - i = 1, j = 1, 2, …, n
  - i = 2, j = 2, 3, …, n
  - …
  - …
  - i = n-1, j = n-1, n
  - i = n, j = n
- There are n(n+1)/2 possible ranges

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

A Brute-force Solution

- Try every possible pair of range [i, j] and compute $A[i]+A[i+1]+\ldots+A[j]$.
- Since we have $\Theta(n^2)$ pairs, it takes $O(n^3)$ time.

A Brute-force Algorithm

```
FIND-MAXIMUM-SUBARRAY1(A)
1.    MaxSum = 0
2.    for i= 1 to n
3.        for j= i to n
4.            ThisSum = 0
5.            for k= i to j
6.                ThisSum = ThisSum + A[ k ]
7.            if ThisSum > MaxSum
8.                MaxSum = ThisSum
9.    return MaxSum
```

- Can you do better?

Actual Running Time

- For $n = 100$, actual time is 0.47 seconds on a particular computer.
- Can use this to estimate time for larger inputs:

  $$T(n) = cn^3$$

  $$T(10n) = c(10n)^3 = 1000cn^3 = 1000T(n)$$

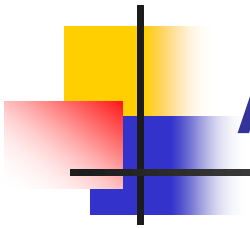
- Inputs size increases by a factor of 10 means that running time increases by a factor of 1,000.
- For $n = 1{,}000$, estimate of running time is 470 seconds. (Actual running time was 449 seconds).
- For $n = 10{,}000$, estimate of running time is 449000 seconds (6 days).

How To Improve

- Remove a loop; not always possible.
- Here it is: innermost loop is unnecessary because it throws away information.
- ThisSum for next j is easily obtained from old value of ThisSum
  - Need $A[i] + A[i+1] + \cdots + A[j-1] + A[j]$
  - Just computed $A[i] + A[i+1] + \cdots + A[j-1]$
  - What we need is $(what\ we\ just\ computed) + A[j]$

A Better Brute-force Algorithm

```
FIND-MAXIMUM-SUBARRAY2(A)
1.    MaxSum = 0
2.    for i= 1 to n
3.        ThisSum = 0
4.        for j= i to n
5.            ThisSum = ThisSum + A[ j ]
6.            if ThisSum > MaxSum
7.                MaxSum = ThisSum
8.    return MaxSum
```



Analysis

- Same logic as before: now the running time is quadratic, or $O(n^2)$.
- As we will see, this algorithm is still usable for inputs in the tens of thousands.
- Recall that the cubic algorithm was not practical for this amount of input.

Actual running time

- For $n = 100$, actual time is 0.011 seconds on the same particular computer.
- Can use this to estimate time for larger inputs:

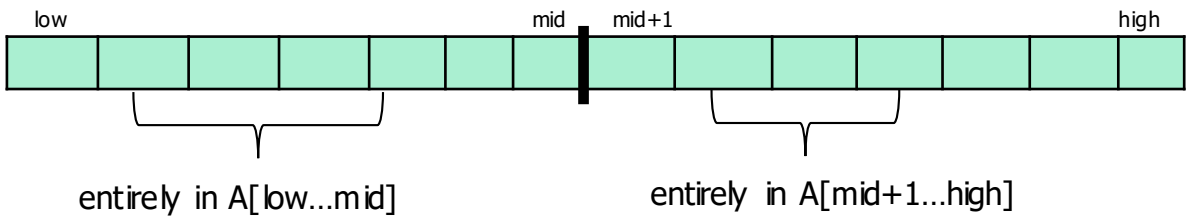
  $$T(n) = cn^2$$

  $$T(10n) = c(10n)^2 = 100cn^2 = 100T(n)$$

- Inputs size increases by a factor of 10 means that running time increases by a factor of 100.
- For $N = 1{,}000$, estimate of running time is 1.11 seconds. (Actual was 1.12 seconds).
- For $N = 10{,}000$, estimate of running time is 111 seconds.

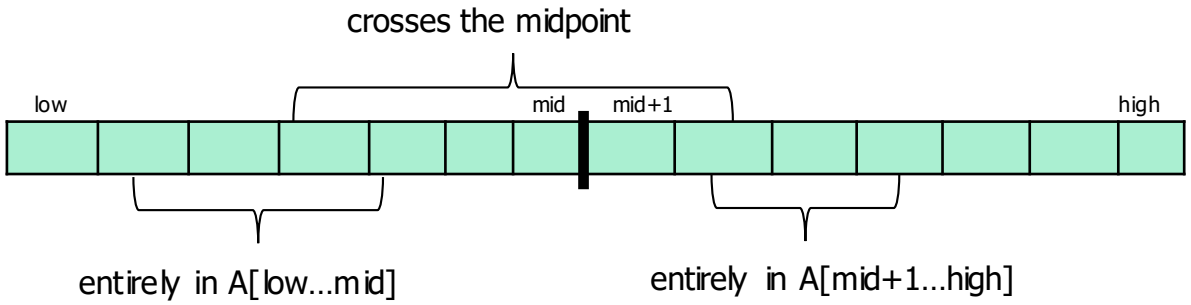
Divide-and-Conquer Algorithm

- The maximum subsequence either
  - lies entirely in the first half
  - lies entirely in the second half
- What is wrong with this?



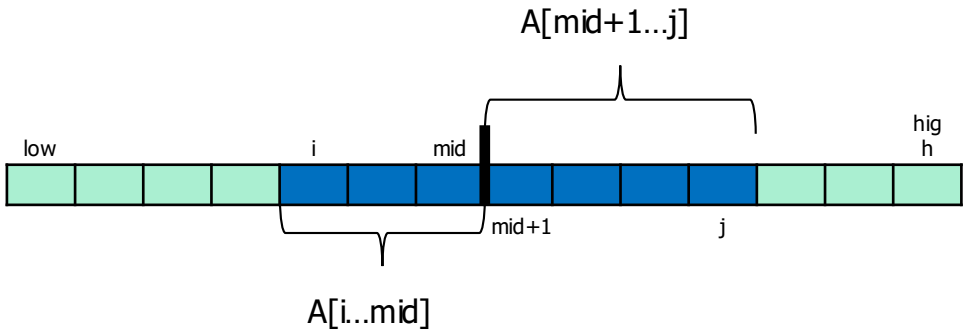
Divide-and-Conquer Algorithm

- The maximum subsequence either
  - lies entirely in the first half
  - lies entirely in the second half
  - starts somewhere in the first half, goes to the last element in the first half, continues at the first element in the second half, ends somewhere in the second half.
- Compute all three possibilities, and use the maximum.
- First two possibilities easily computed recursively.



Computing the Third Case

- Easily done with two loops.
- For maximum sum that starts in the first half and extends to the last element in the first half, use a right-to-left scan starting at the last element in the first half.
- For the other maximum sum, do a left-to-right scan, starting at the first element in the first half.

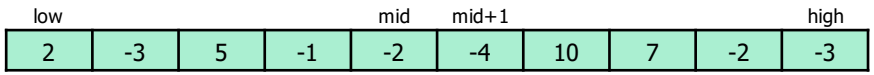


Analysis

- Let $T(n)$ = the time for an algorithm to solve a problem of size N.
- Then $T(1) = 1$ (1 will be the quantum time unit; remember that constants don't matter).
- $T(n) = 2T\left(\frac{n}{2}\right) + n$
  - Two recursive calls, each of size $n/2$. The time to solve each recursive call is $T(n/2)$ by the above definition
  - Case three takes $O(n)$ time

A Divide-and-Conquer Algorithm

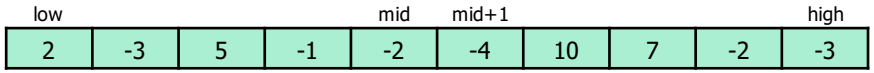
```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
1.    leftSum = -∞
2.    sum = 0
3.    for i = mid downto low
4.        sum = sum + A[i]
5.        if sum > leftSum
6.            leftSum = sum
7.    rightSum = -∞
8.    sum = 0
9.    for i = mid+1 to high
10.       sum = sum + A[i]
11.       if sum > rightSum
12.           rightSum = sum
13.   return leftSum+rightSum
```

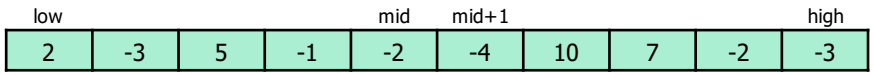
leftSum = -∞
sum = 0



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
1.    leftSum = -∞
2.    sum = 0
3.    for i = mid downto low
4.        sum = sum + A[i]
5.        if sum > leftSum
6.            leftSum = sum
7.    rightSum = -∞
8.    sum = 0
9.    for i = mid+1 to high
10.       sum = sum + A[i]
11.       if sum > rightSum
12.           rightSum = sum
13.   return leftSum+rightSum
```

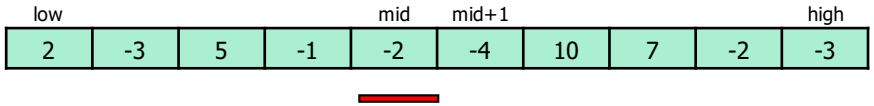
leftSum = -∞
sum = 0



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

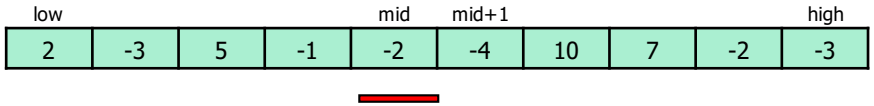
leftSum = -∞
sum = -2



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

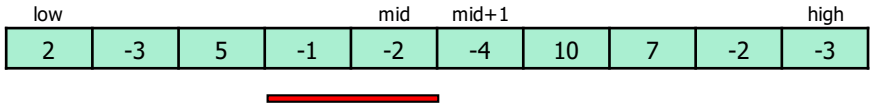
leftSum = -2
sum = -2



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

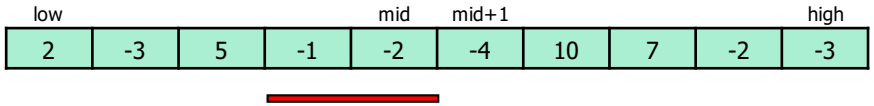
leftSum = -2
sum = -3



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

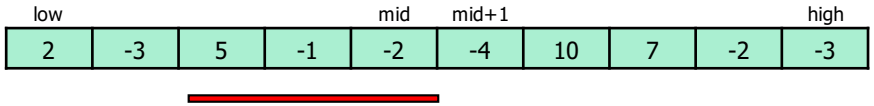
leftSum = -2
sum = -3



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

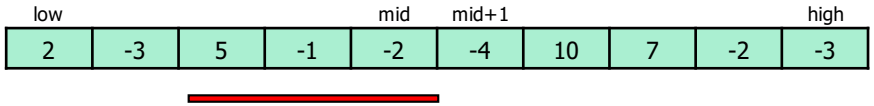
leftSum = -2
sum = 2



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

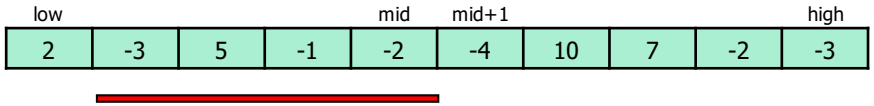
leftSum = 2
sum = 2



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

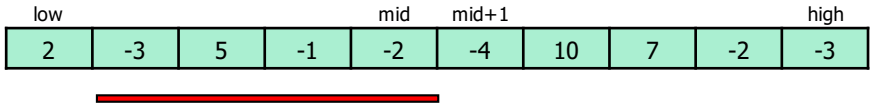
leftSum = 2
sum = -1



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

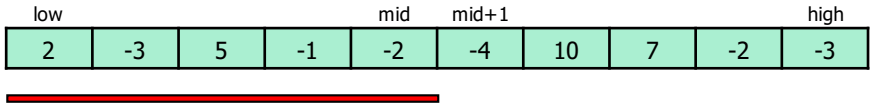
leftSum = 2
sum = -1



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

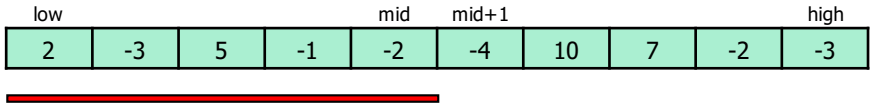
leftSum = 2
sum = 1



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

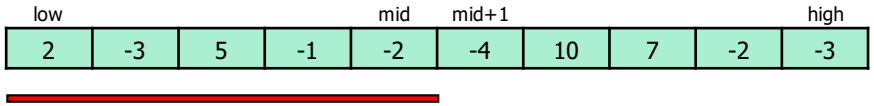
leftSum = 2
sum = 1



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

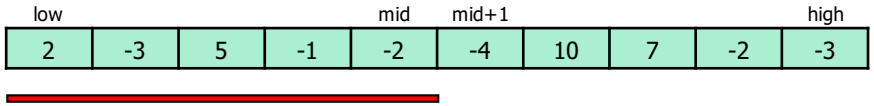
leftSum = 2
rightSum = -∞
sum = 0



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

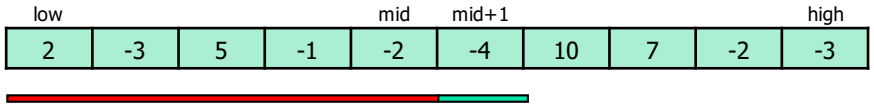
leftSum = 2
rightSum = -∞
sum = 0



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

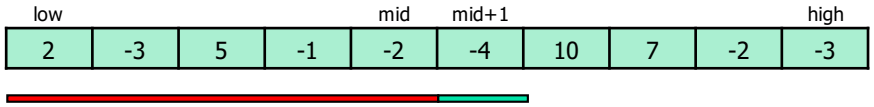
leftSum = 2
rightSum = -∞
sum = -4



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

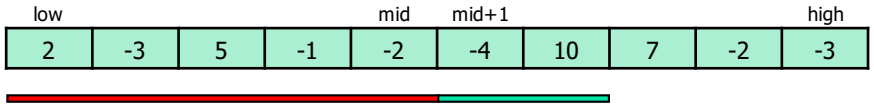
leftSum = 2
rightSum = -4
sum = -4



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

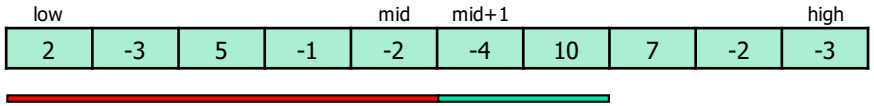
leftSum = 2
rightSum = -4
sum = 6



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

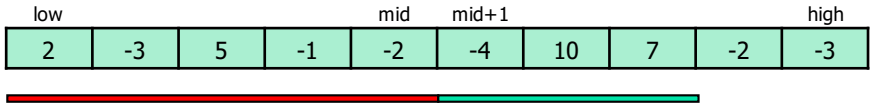
leftSum = 2
rightSum = 6
sum = 6



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

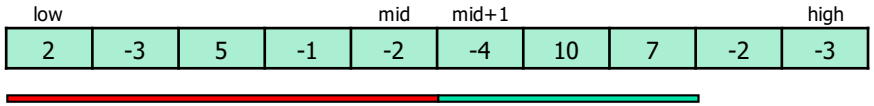
leftSum = 2
rightSum = 6
sum = 13



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

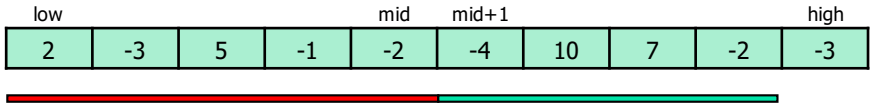
leftSum = 2
rightSum = 13
sum = 13



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

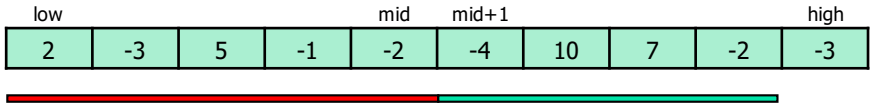
leftSum = 2
rightSum = 13
sum = 11



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

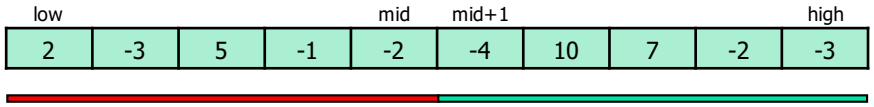
leftSum = 2
rightSum = 13
sum = 11



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

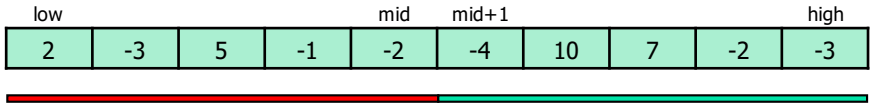
leftSum = 2
rightSum = 13
sum = 8



A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

leftSum = 2
rightSum = 13
sum = 8

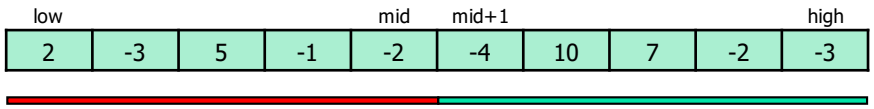


A Divide-and-Conquer Algorithm

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1.    leftSum = -∞
 2.    sum = 0
 3.    for i = mid downto low
 4.        sum = sum + A[i]
 5.        if sum > leftSum
 6.            leftSum = sum
 7.    rightSum = -∞
 8.    sum = 0
 9.    for i = mid+1 to high
10.        sum = sum + A[i]
11.        if sum > rightSum
12.            rightSum = sum
13.    return leftSum+rightSum
```

leftSum = 2
rightSum = 13
sum = 8

return 15



A Divide-and-Conquer Algorithm

- 앞에서 만든 FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high) 함수를 이용해서 FIND-MAXIMUM-SUBARRAY3(A, low, high)를 divide and conqer로 작성해 보세요.
  - maximum range의 array 상의 왼쪽인덱스, 오른쪽 인덱스, maximum range sum 이 세가지를 리턴하세요.

A Divide-and-Conquer Algorithm

```
FIND-MAXIMUM-SUBARRAY3(A, low, high)
 1.   if high == low
 2.      return (low, high, A[low] )
 3.   else  mid = (low +high)/2
 4.      (leftLow, leftHigh, leftSum) =  FIND-MAXIMUM-SUBARRAY3(A, low, mid)
 5.      (rightLow, rightHigh, rightSum) = FIND-MAXIMUM-SUBARRAY3(A, mid+1, high)
 6.      (crossLow, crossHigh, crossSum) = FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7.      if leftSum >= rightSum and leftSum >= crossSum
 8.         return (leftLow, leftHigh, leftSum)
 9.      if rightSum >= leftSum and rightSum >= crossSum
10.         return (rightLow, rightHigh, rightSum)
11.      else return (crossLow, crossHigh, crossSum)
```

Incremental Algorithm

- Kadane's Algorithm
- If we know
  - the maximum subarray sum ending at position A[i] (Call it ThisSum)
  - the maximum subarray sum for the range [1, i] (Call it MaxSum)
- What is the maximum subarray sum for the range [i, i+1]?
  - max(MaxSum, A[i+1], ThisSum + A[i+1])

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

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

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 0
ThisSum = 0

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

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 0
ThisSum = 0

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

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 0
ThisSum = 2
j = 1

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

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 2
ThisSum = 2
j = 1

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

range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.       ThisSum = ThisSum + A[ j ]
5.       if ThisSum > MaxSum
6.          MaxSum = ThisSum
7.       else if ThisSum < 0
8.          ThisSum = 0
9.    return MaxSum
```

MaxSum = 2
ThisSum = -1
j = 2

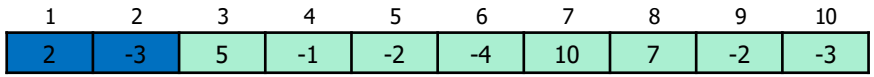
1	2	3	4	5	6	7	8	9	10
2	-3	5	-1	-2	-4	10	7	-2	-3

range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 2
ThisSum = -1
j = 2

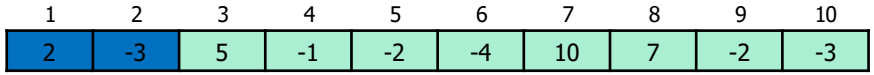


range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 2
ThisSum = 0
j = 2

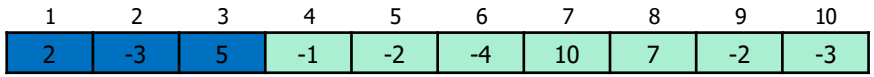


range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.       ThisSum = ThisSum + A[ j ]
5.       if ThisSum > MaxSum
6.          MaxSum = ThisSum
7.       else if ThisSum < 0
8.          ThisSum = 0
9.    return MaxSum
```

MaxSum = 2
ThisSum = 5
j = 3

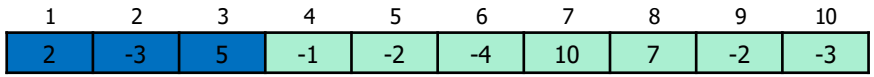


range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 5
ThisSum = 5
j = 3

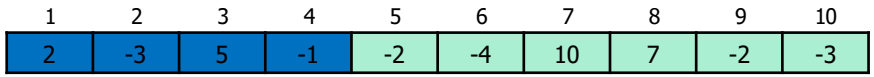


range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.       ThisSum = ThisSum + A[ j ]
5.       if ThisSum > MaxSum
6.          MaxSum = ThisSum
7.       else if ThisSum < 0
8.          ThisSum = 0
9.    return MaxSum
```

MaxSum = 5
ThisSum = 4
j = 4



range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 5
ThisSum = 4
j = 4

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

range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 5
ThisSum = 4
j = 4

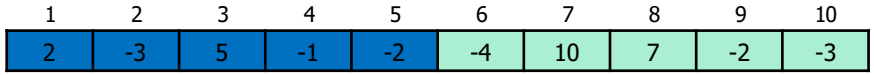
1	2	3	4	5	6	7	8	9	10
2	-3	5	-1	-2	-4	10	7	-2	-3

range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.       ThisSum = ThisSum + A[ j ]
5.       if ThisSum > MaxSum
6.          MaxSum = ThisSum
7.       else if ThisSum < 0
8.          ThisSum = 0
9.    return MaxSum
```

MaxSum = 5
ThisSum = 2
j = 5

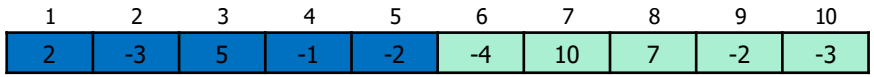


range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 5
ThisSum = 2
j = 5



range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 5
ThisSum = 2
j = 5

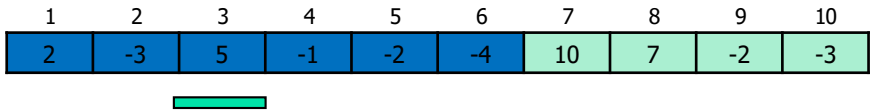
1	2	3	4	5	6	7	8	9	10
2	-3	5	-1	-2	-4	10	7	-2	-3

range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.       ThisSum = ThisSum + A[ j ]
5.       if ThisSum > MaxSum
6.          MaxSum = ThisSum
7.       else if ThisSum < 0
8.          ThisSum = 0
9.    return MaxSum
```

MaxSum = 5
ThisSum = -2
j = 6

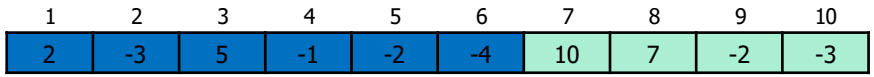


range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 5
ThisSum = -2
j = 6



range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 5
ThisSum = 0
j = 6

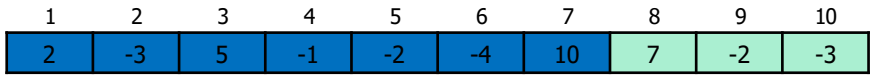
1	2	3	4	5	6	7	8	9	10
2	-3	5	-1	-2	-4	10	7	-2	-3

range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.       ThisSum = ThisSum + A[ j ]
5.       if ThisSum > MaxSum
6.          MaxSum = ThisSum
7.       else if ThisSum < 0
8.          ThisSum = 0
9.    return MaxSum
```

MaxSum = 5
ThisSum = 10
j = 7

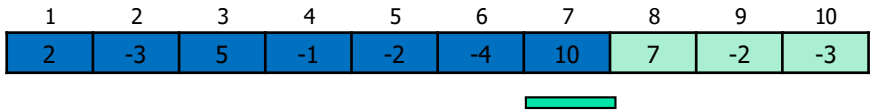


range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 10
ThisSum = 10
j = 7

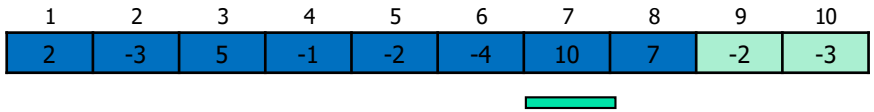


range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.       ThisSum = ThisSum + A[ j ]
5.       if ThisSum > MaxSum
6.          MaxSum = ThisSum
7.       else if ThisSum < 0
8.          ThisSum = 0
9.    return MaxSum
```

MaxSum = 10
ThisSum = 17
j = 8

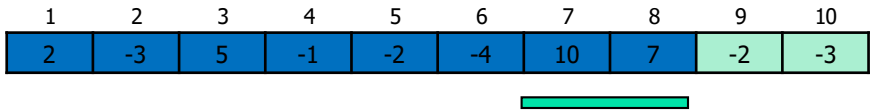


range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 17
ThisSum = 17
j = 8

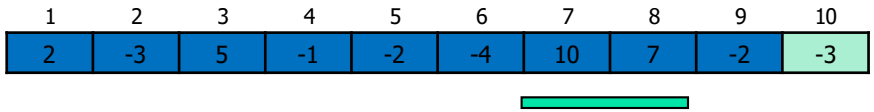


range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.       ThisSum = ThisSum + A[ j ]
5.       if ThisSum > MaxSum
6.          MaxSum = ThisSum
7.       else if ThisSum < 0
8.          ThisSum = 0
9.    return MaxSum
```

MaxSum = 17
ThisSum = 15
j = 9

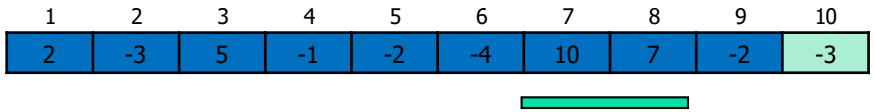


range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 17
ThisSum = 15
j = 9

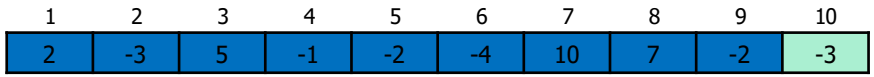


range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 17
ThisSum = 15
j = 9



range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 17
ThisSum = 12
j = 10

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

range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 17
ThisSum = 12
j = 10

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

range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 17
ThisSum = 12
j = 10

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

range of MaxSum

A Linear-time Algorithm

```
FIND-MAXIMUM-SUBARRAY4(A)
1.    MaxSum = 0
2.    ThisSum = 0
3.    for j = 1 to n
4.        ThisSum = ThisSum + A[ j ]
5.        if ThisSum > MaxSum
6.            MaxSum = ThisSum
7.        else if ThisSum < 0
8.            ThisSum = 0
9.    return MaxSum
```

MaxSum = 17
ThisSum = 12
j = 11

return 17

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

range of MaxSum

A Linear-time Algorithm

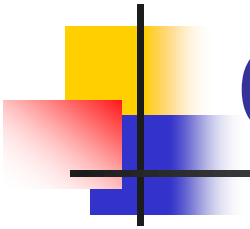
- Linear time (i.e., $O(n)$) algorithm would be best
- Running time is proportional to amount of input
- It makes only one pass through the data
- If the array is on a disk or is being transmitted over the Internet, it can be read sequentially, and there is no need to store any part of it in main memory
- At any point in time, the algorithm can correctly give an answer to the subsequence problem for the data it has already read: We call online algorithm

Any Question?

Introduction to Algorithms (Chapter 6: Heap Sort)

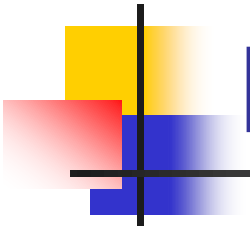
Kyuseok Shim

Electrical and Computer Engineering
Seoul National University



Outline

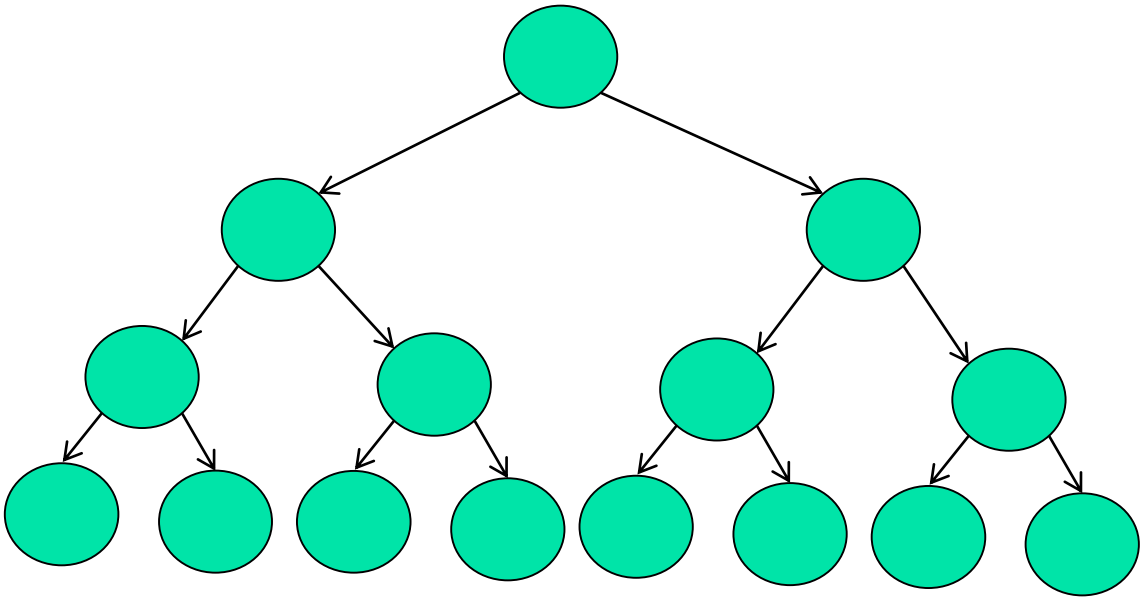
- This chapter introduces a sorting algorithm called the heapsort.



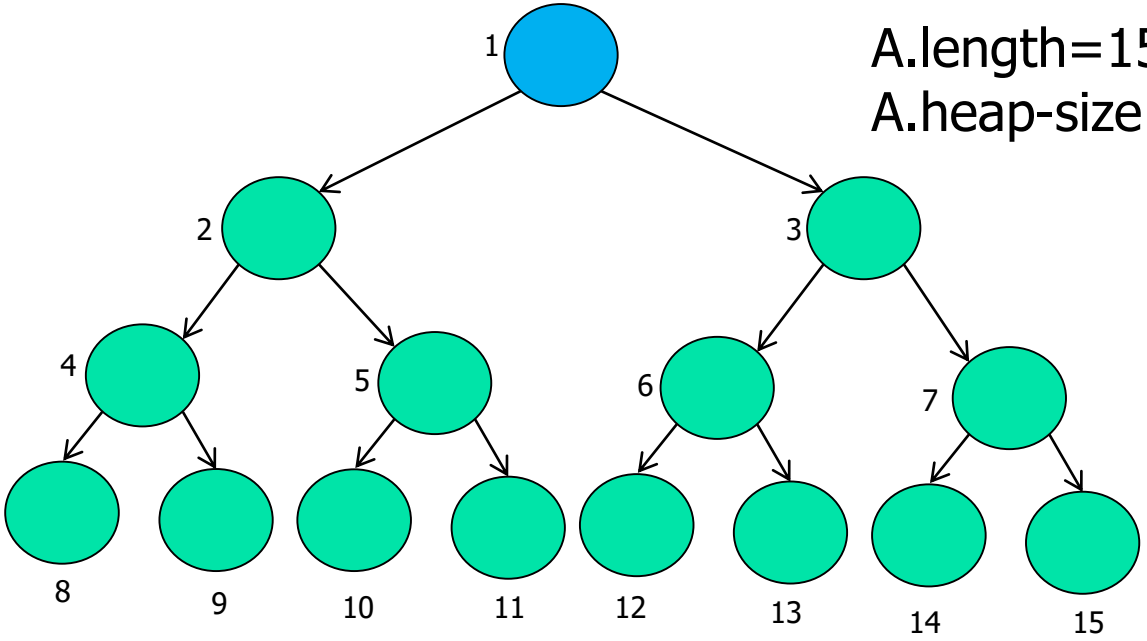
Heapsort

- Compared to insertion sort,
  - It sorts in place
  - In contrast, its running time is O(n lg n)
- Compared to merge sort,
  - Running time is the same as O(n lg n)
  - In contrast, it does not require O(n) additional space

A Complete Binary Tree

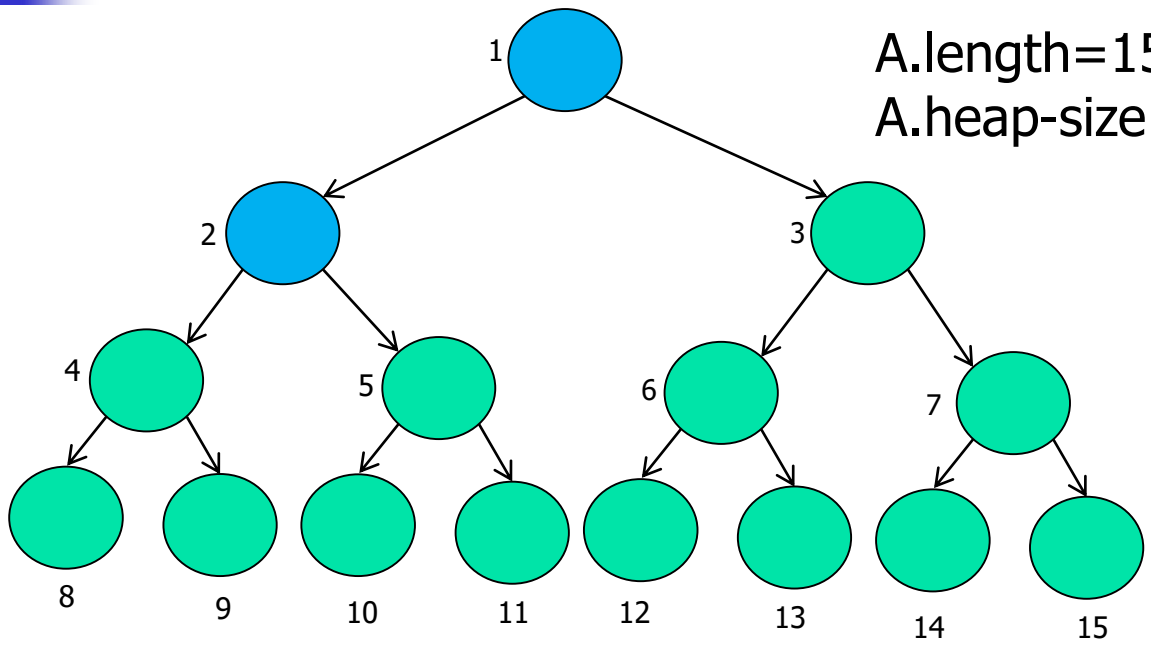


A Possible Heap



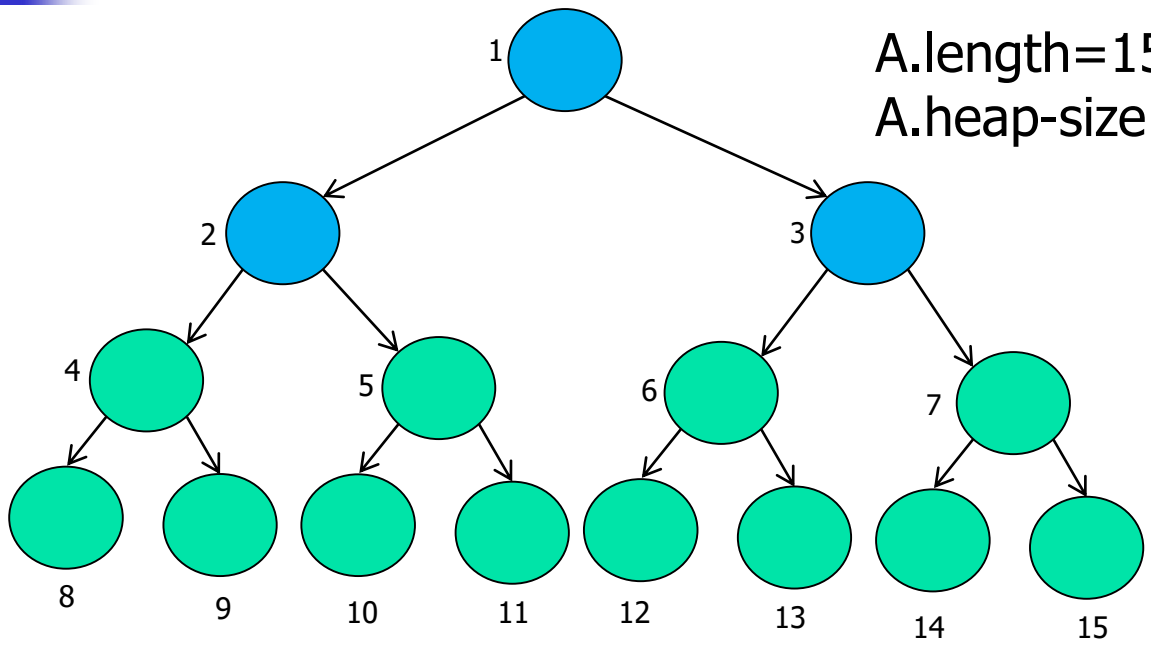
A.length=15
A.heap-size=1

A Possible Heap



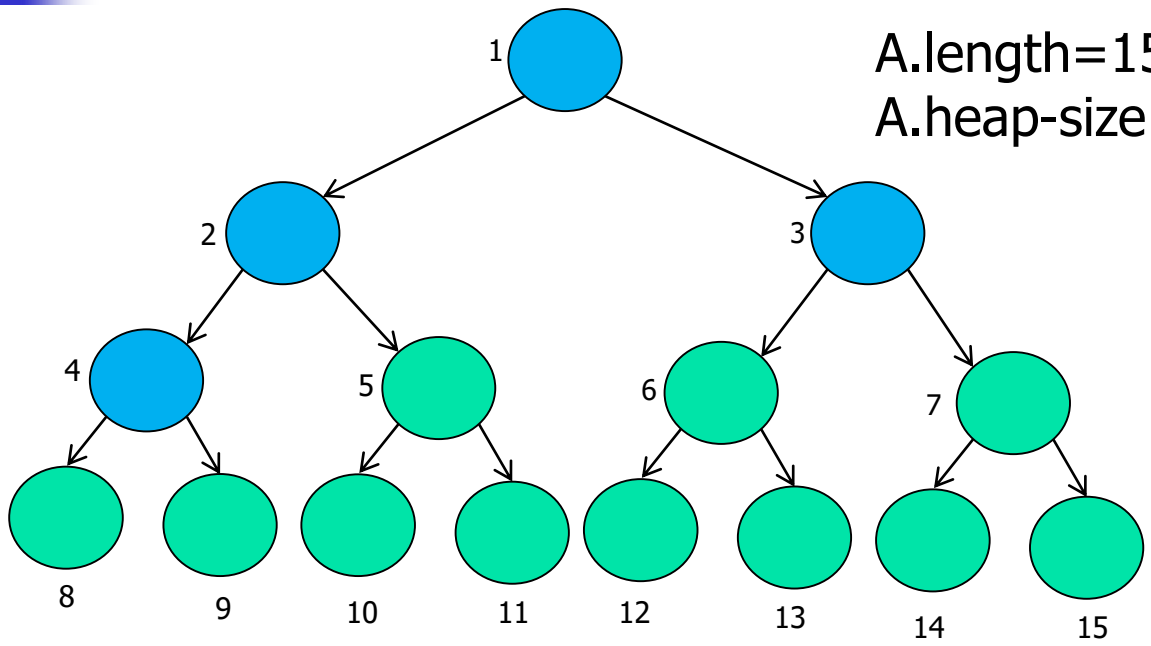
A.length=15
A.heap-size=2

A Possible Heap



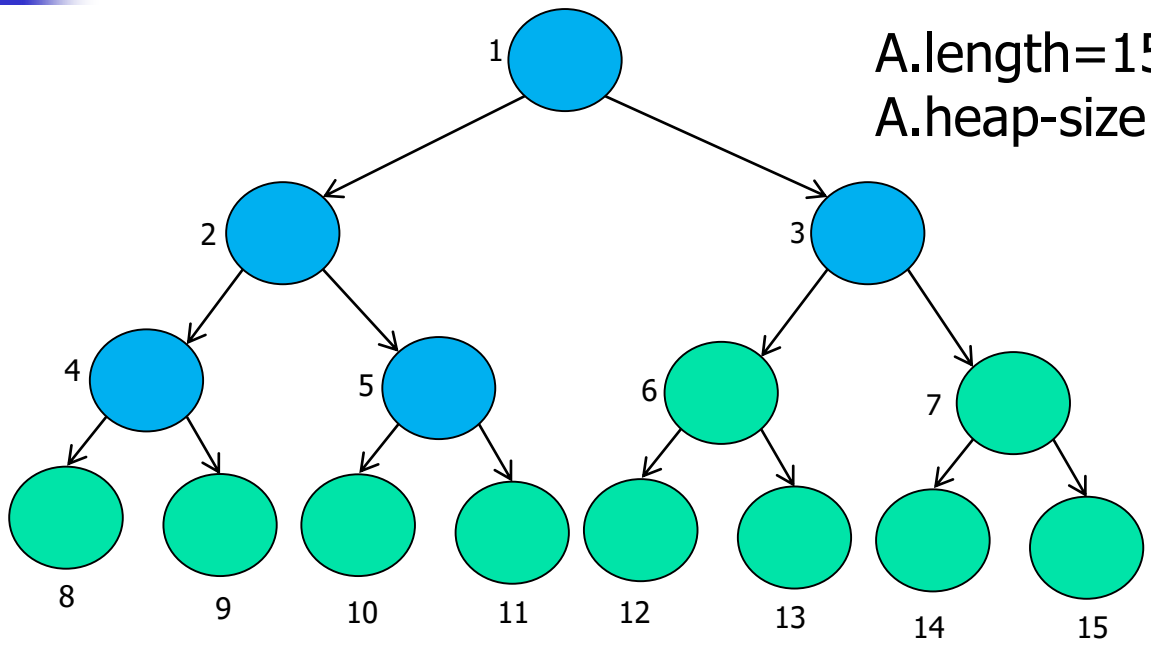
A.length=15
A.heap-size=3

A Possible Heap



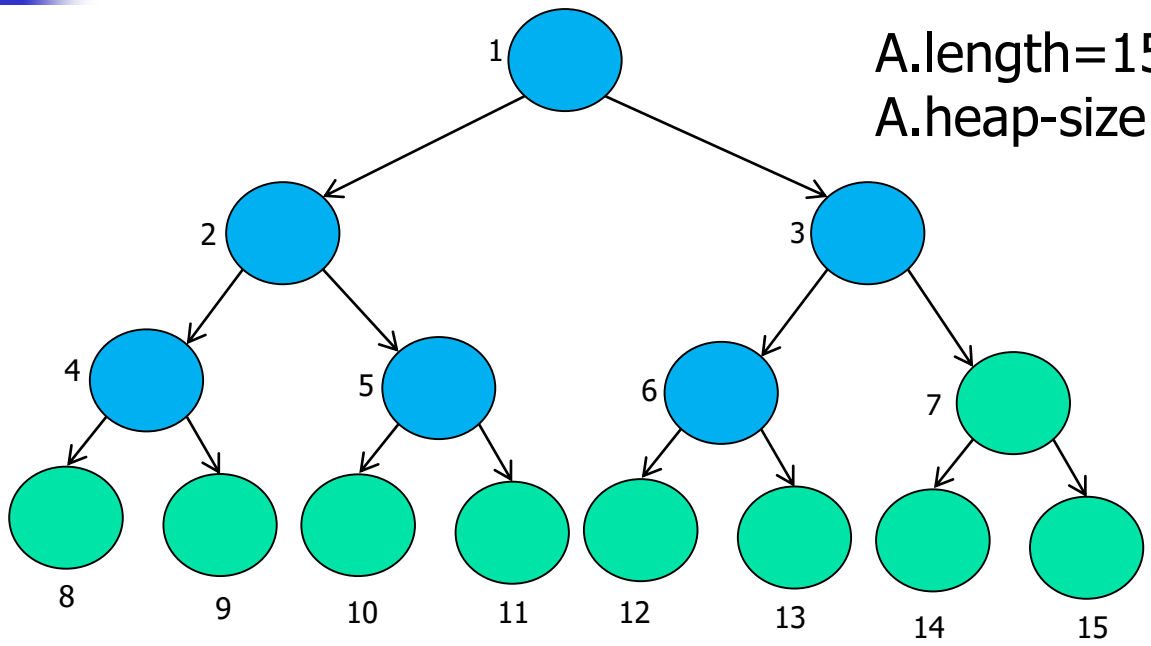
A.length=15
A.heap-size=4

A Possible Heap



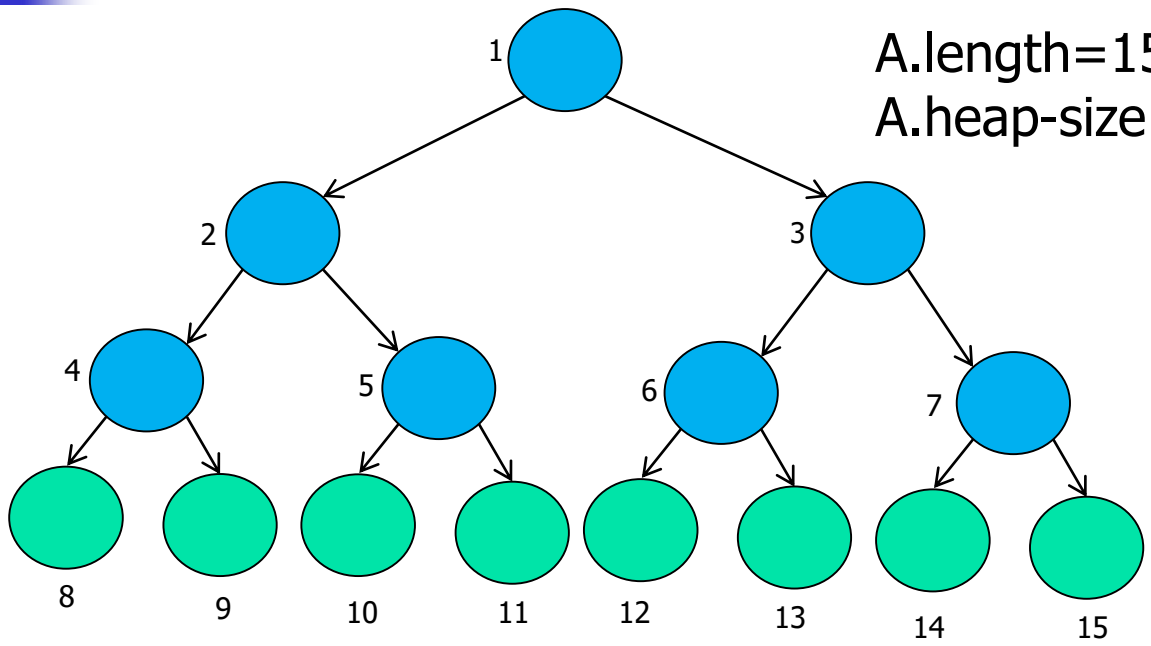
A.length=15
A.heap-size=5

A Possible Heap



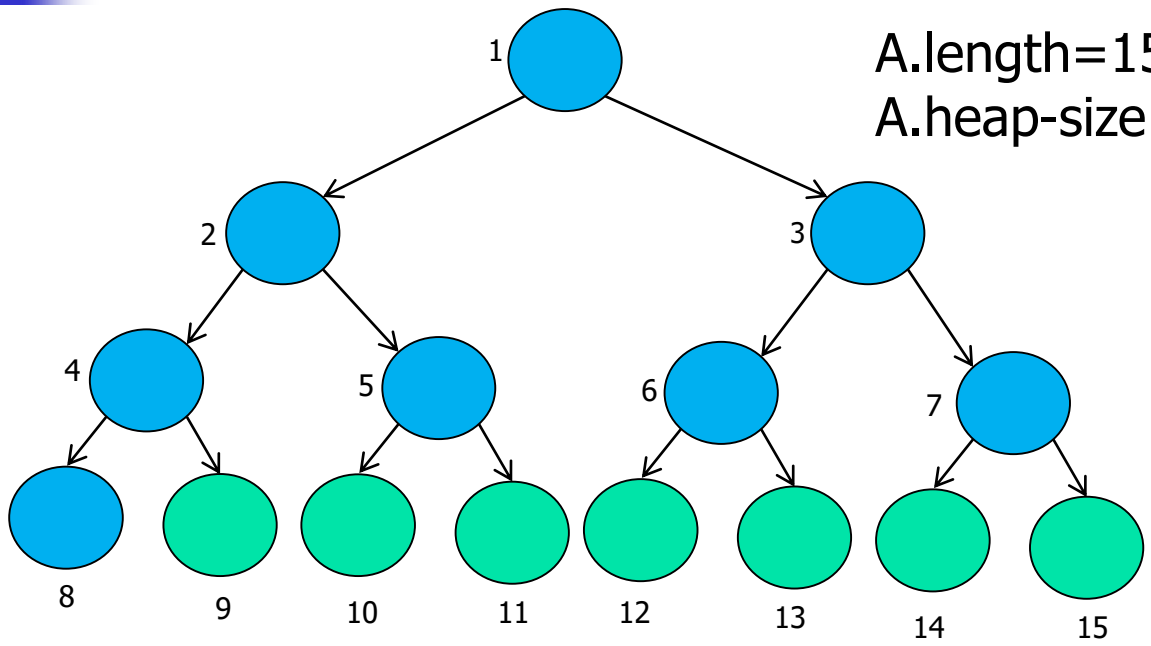
A.length=15
A.heap-size=6

A Possible Heap

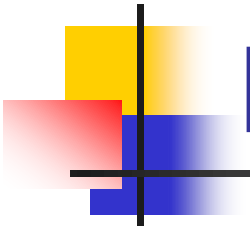


A.length=15
A.heap-size=7

A Possible Heap



A.length=15
A.heap-size=8

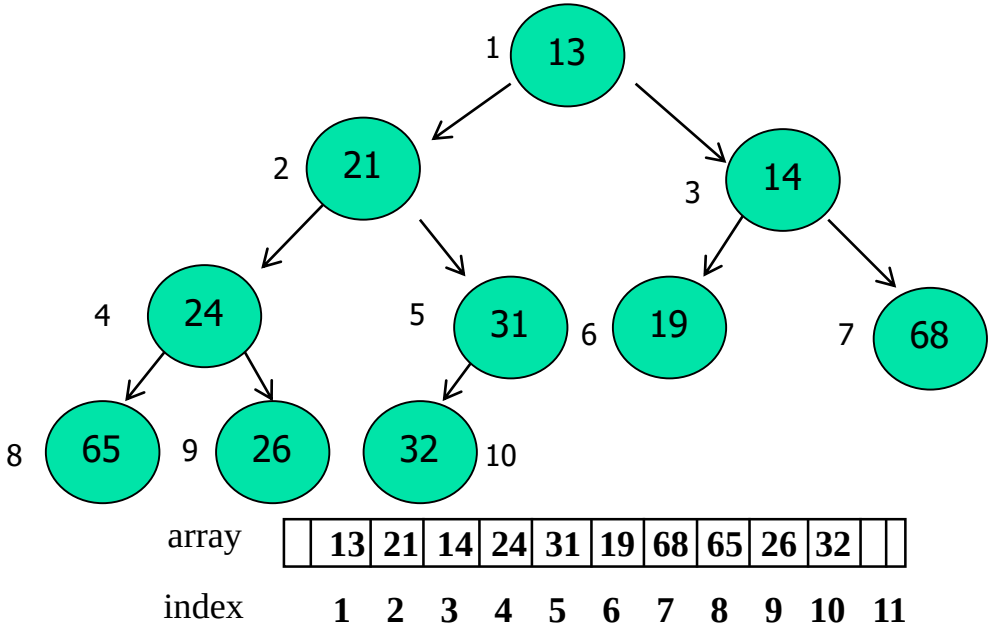


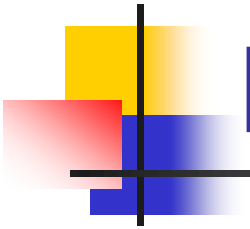
Heaps

- The binary heap is an array that we can view as a nearly complete binary tree.
- Each node of the tree corresponds to an element of the array.
- The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.
- An array A that represents a heap is an object with two attributes
  - A.length: the number of elements in the array
  - A.heap-size: the number of elements in the heap
- Only the elements in A[1…A.heap-size], where 0 ≤ A.heap-size ≤ A.length, are valid elements of the heap.
- The array element A[1] is the root of the tree
- Given the index i, the indices of its parent, left child and right child are
  - LEFT(i)=2i (if 2i≤A.heap-size)
  - RIGHT(i)=2i+1 (if 2i+1≤A.heap-size)
  - PARENT(i)=$\lfloor i/2 \rfloor$

Min Heap

- For every node i other than the root, A[PARENT(i)] ≤ A[i]





Heap

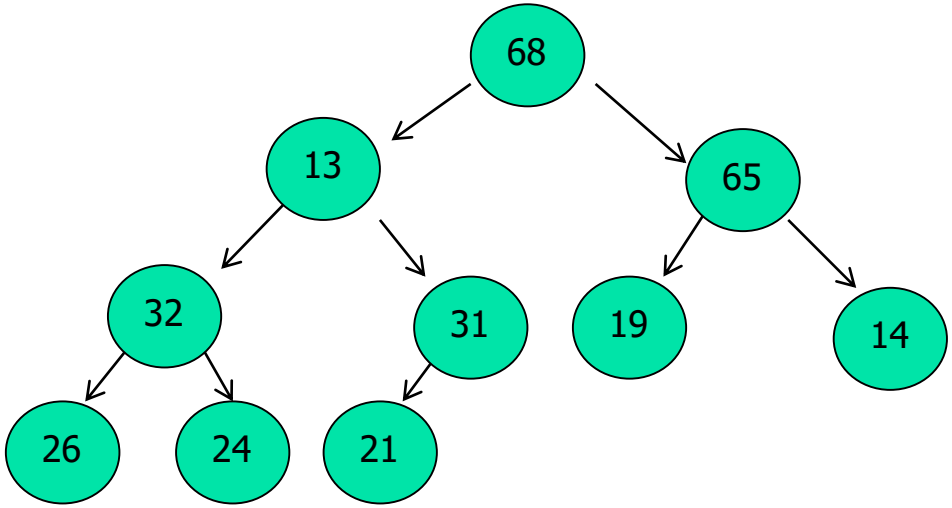
- The height of a node in a heap is the number of edges on the longest simple downward path from the node to a leaf.
- The height of the heap is the height of its root.
- Since a heap of n elements is based on a complete binary tree, its height is O(lg n)

MAX-HEAP

- The binary trees rooted at LEFT(i) and RIGHT(i) are max heaps and A[i] might violate the max-heap property.
- MAX-HEAPIFY lets the value at A[i] float down so that the subtree rooted at index i becomes a max-heap.
- MAX-HEAPIFY(A, i)

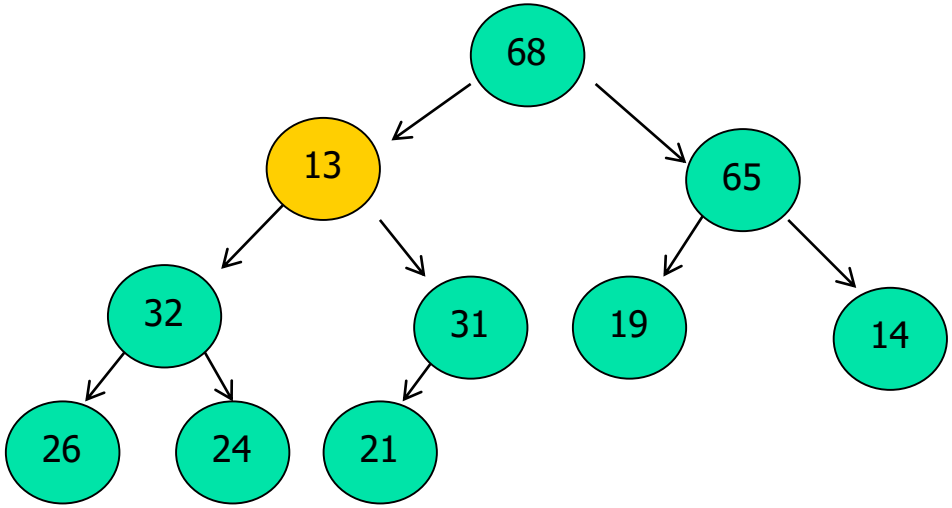
```
1.    L  = Left(i)
2.    R  = Right(i)
3.    if L ≤ A.heap-size and A[L] > A[i]
4.       largest = L
5.    else largest  = i
6.    if R ≤ A.heap-size and A[R] > A[largest]
7.       then largest = R
8.    if largest ≠ i
9.       exchange A[i] with A[largest]
10.      MAX-HEAPIFY(A, largest)
```

Action of MAX-HEAPIFY(A,2)



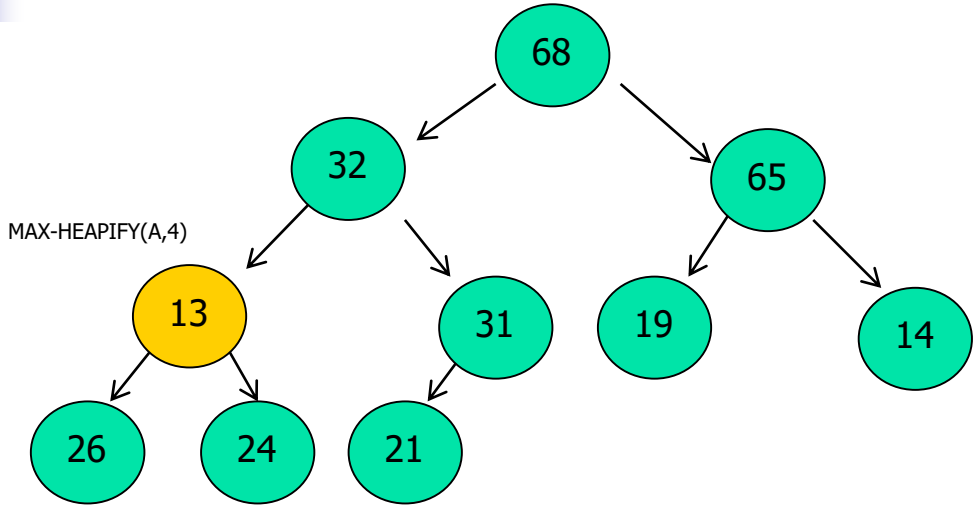
array		68	13	65	32	31	19	14	26	24	21	
index	0	1	2	3	4	5	6	7	8	9	10	11

Action of MAX-HEAPIFY(A,2)



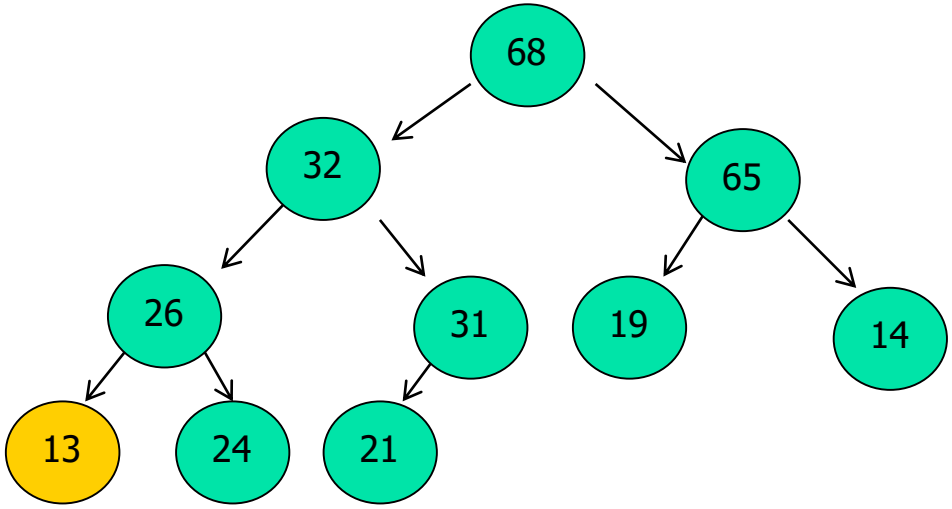
array		68	13	65	32	31	19	14	26	24	21	
index	0	1	2	3	4	5	6	7	8	9	10	11

Action of MAX-HEAPIFY(A,2)



array		68	32	65	13	31	19	14	26	24	21	
index	0	1	2	3	4	5	6	7	8	9	10	11

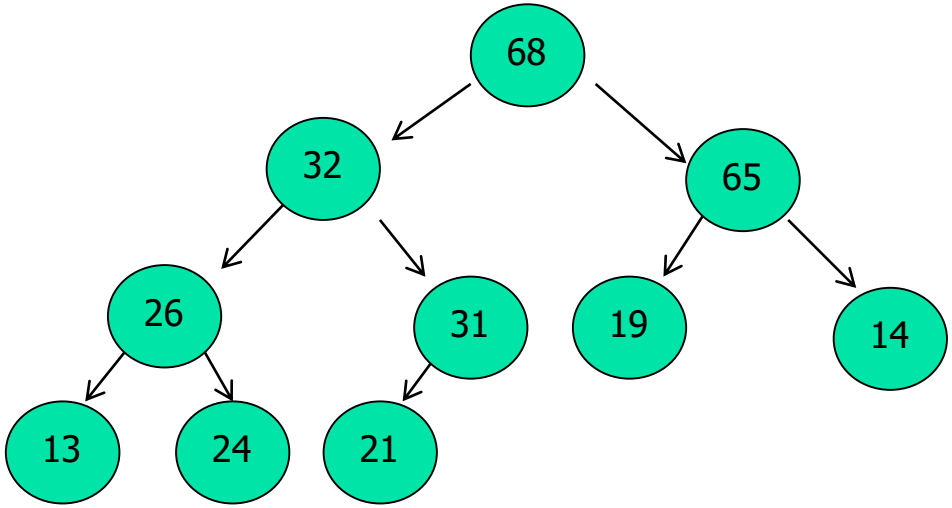
Action of MAX-HEAPIFY(A,2)



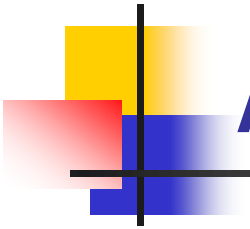
MAX-HEAPIFY(A,8)

array		68	32	65	26	31	19	14	13	24	21	
index	0	1	2	3	4	5	6	7	8	9	10	11

Action of MAX-HEAPIFY(A,2)



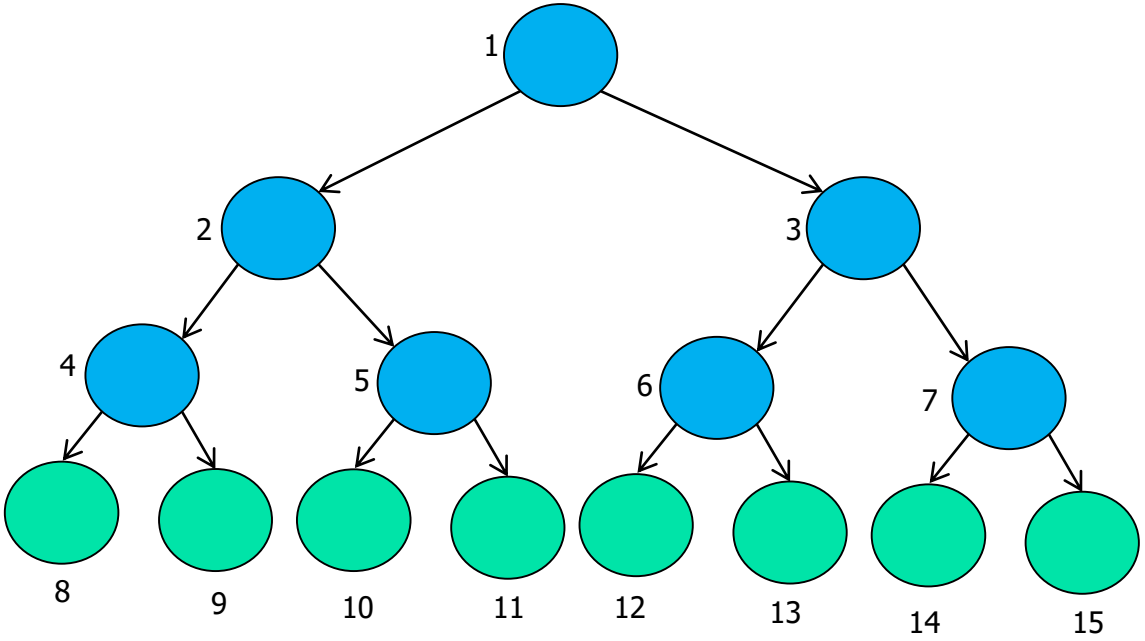
array		68	32	65	13	31	19	14	26	24	21	
index	0	1	2	3	4	5	6	7	8	9	10	11



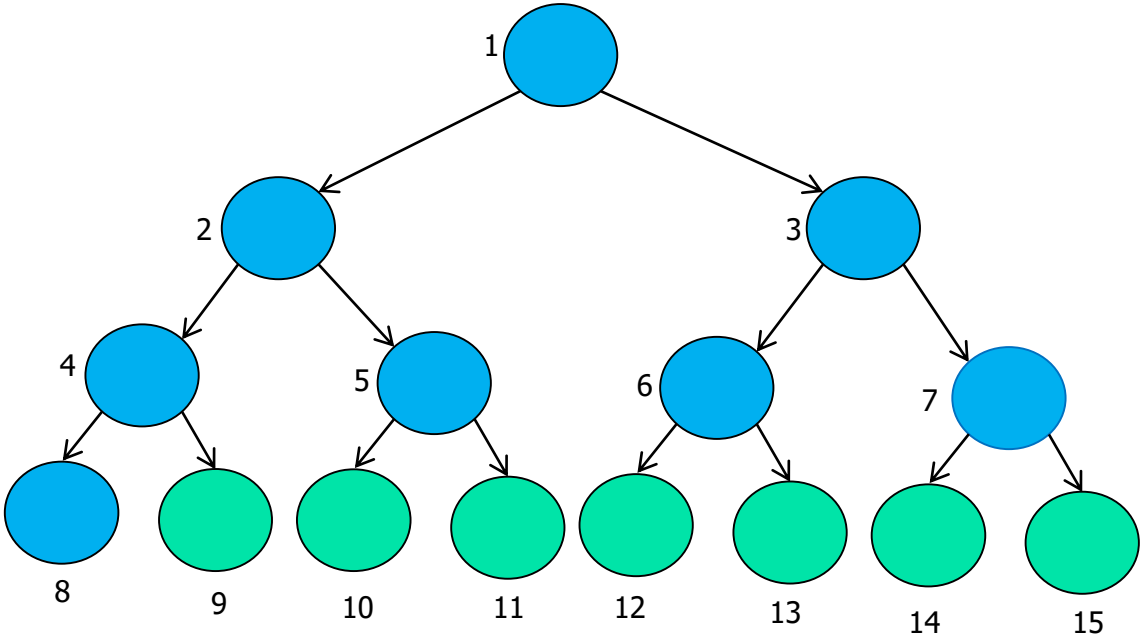
Analysis of MAX-HEAPIFY

- For a subtree of size n rooted at a given node i, the size of the subtree rooted at a child of node i becomes maximum when the bottom level of the tree is exactly half full
  - In other words, the size of the subtree rooted at one of the children of node i is at most 2n/3

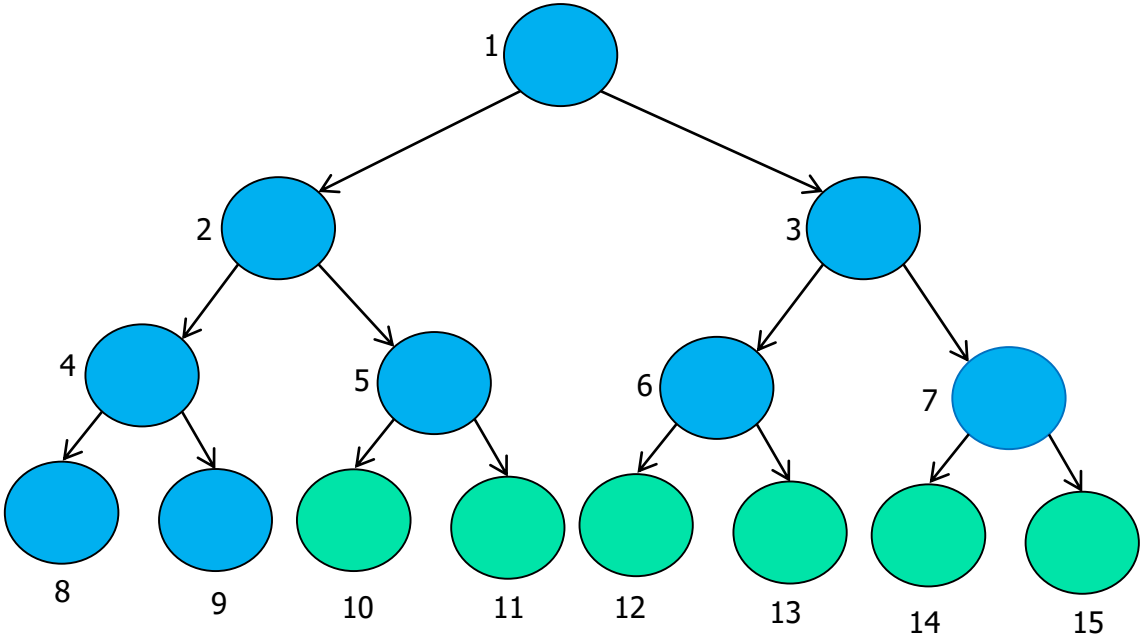
A Possible Heap



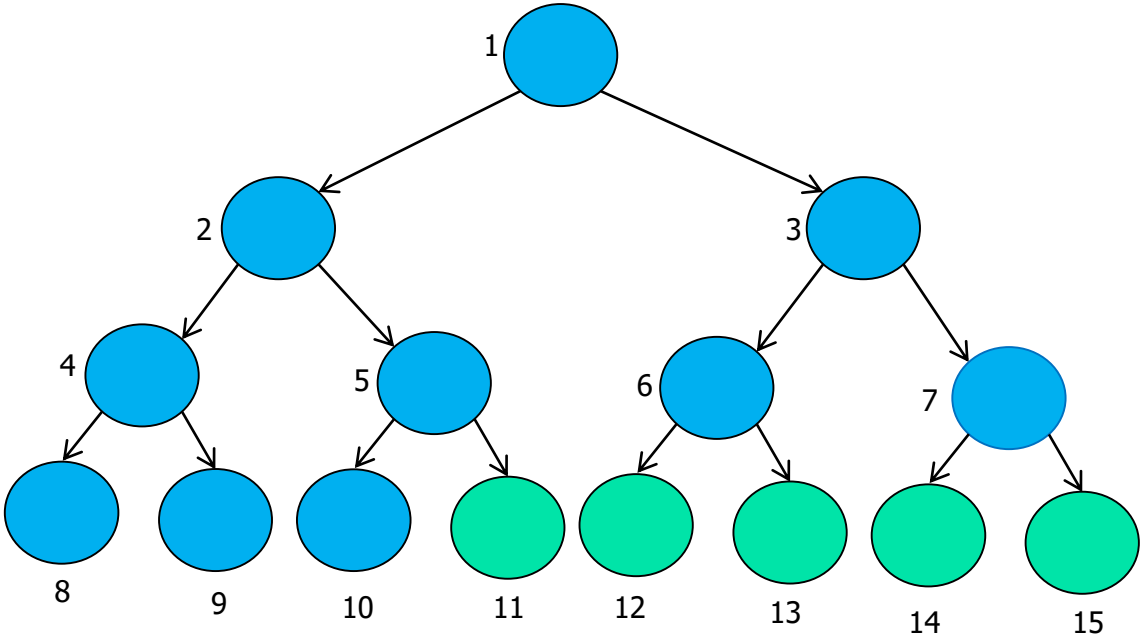
A Possible Heap



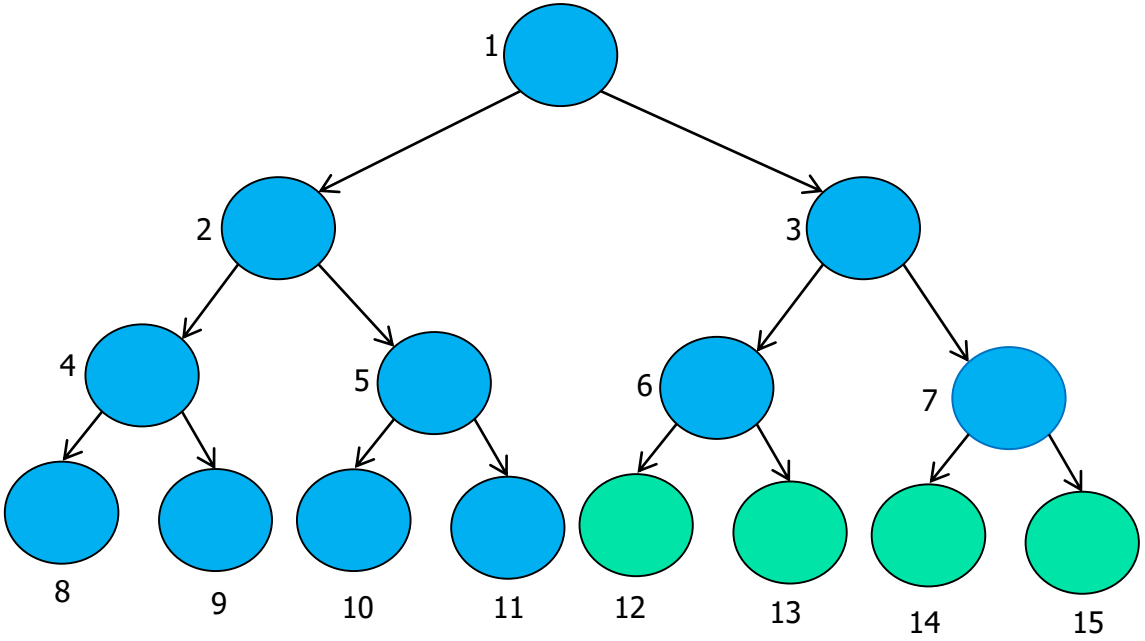
A Possible Heap



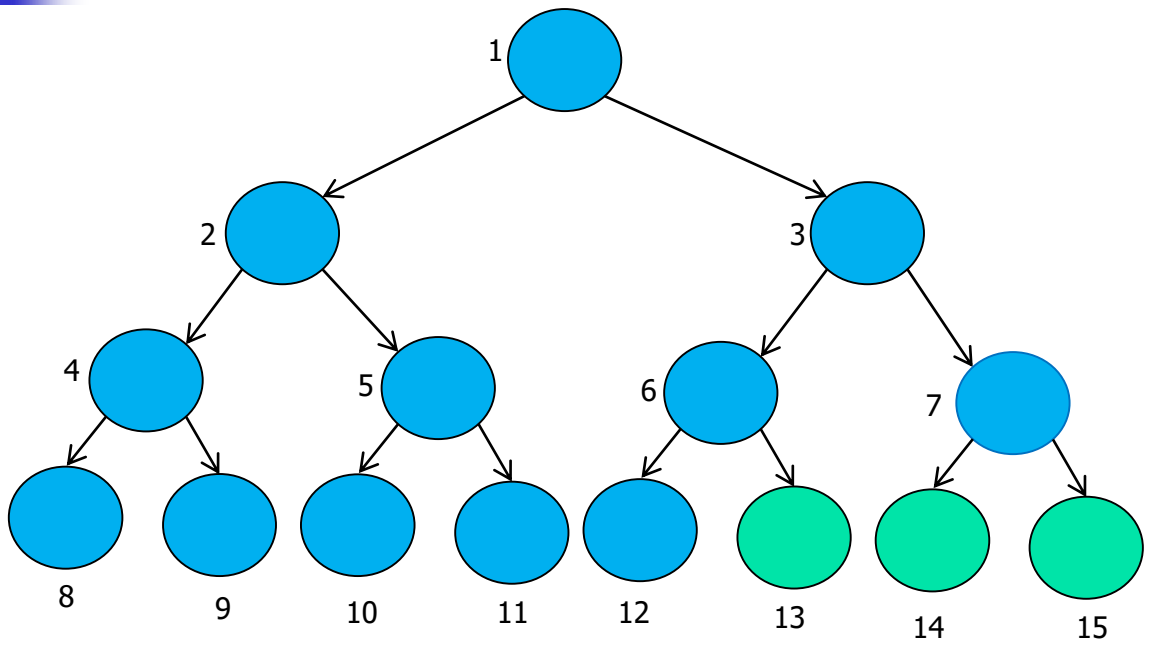
A Possible Heap



A Possible Heap



A Possible Heap



Analysis of MAX-HEAPIFY

- For a subtree of size n rooted at a given node i, the size of the subtree rooted at a child of node i becomes maximum when the bottom level of the tree is exactly half full
  - In other words, the size of the subtree rooted at one of the children of node i is at most 2n/3
- The running time on a subtree of size n rooted at a given node i consist of
  - $\Theta(1)$ to fix up the relationship among the elements A[i], A[LEFT(i)] and A[RIGHT(i)]
  - T(2n/3) to recursively process on a subtree rooted at one of the children of node i
- Thus, we have $T(n) \leq T(2n/3) + \Theta(1)$
  - $T(n) = O(\lg n)$

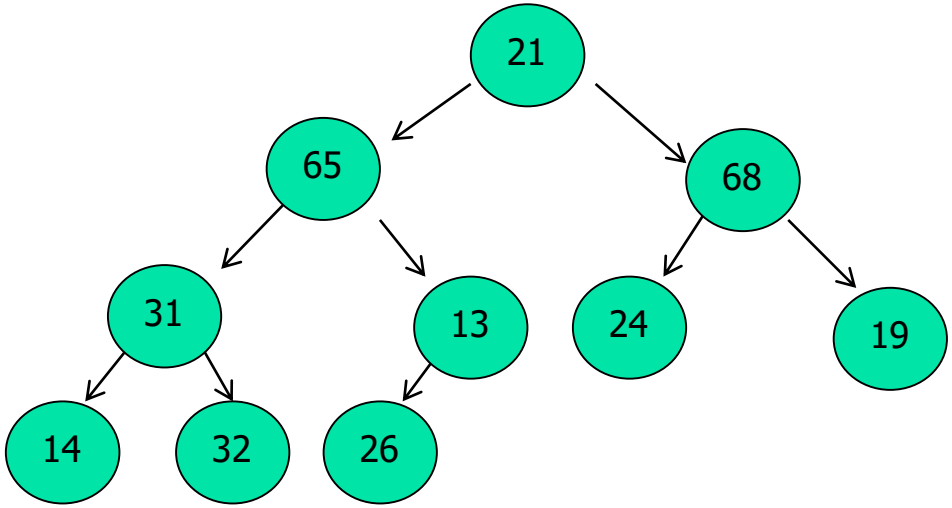
BUILD-MAX-HEAP(A)

- We can use MAX-HEAPIFY in a bottom-up manner to convert array A[1..n], where n = A.length, into a max-heap

BUILD-MAX-HEAP(A)

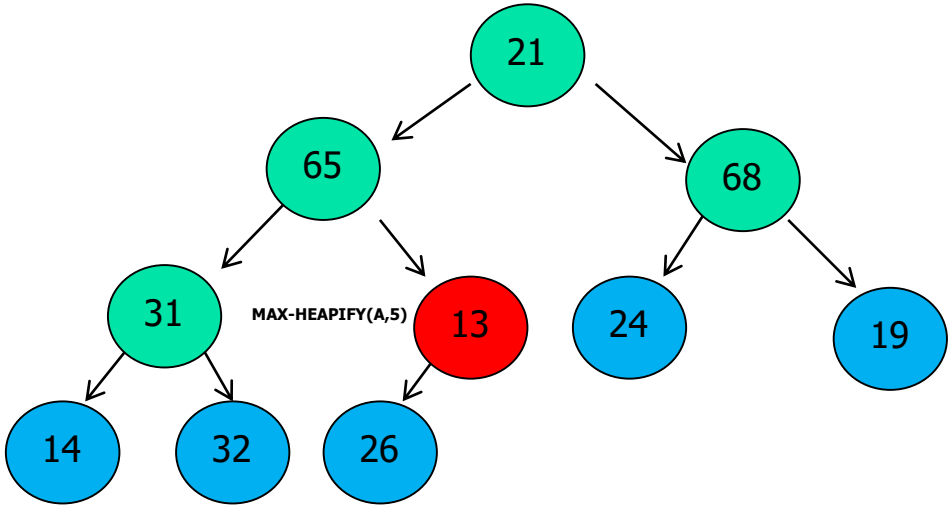
1. A.heap-size = A.length
2. **for** i = $\lfloor A.length/2 \rfloor$ **downto** 1
3. MAX-HEAPIFY(A, i)

Operation of BUILD-MAX-HEAP(A)



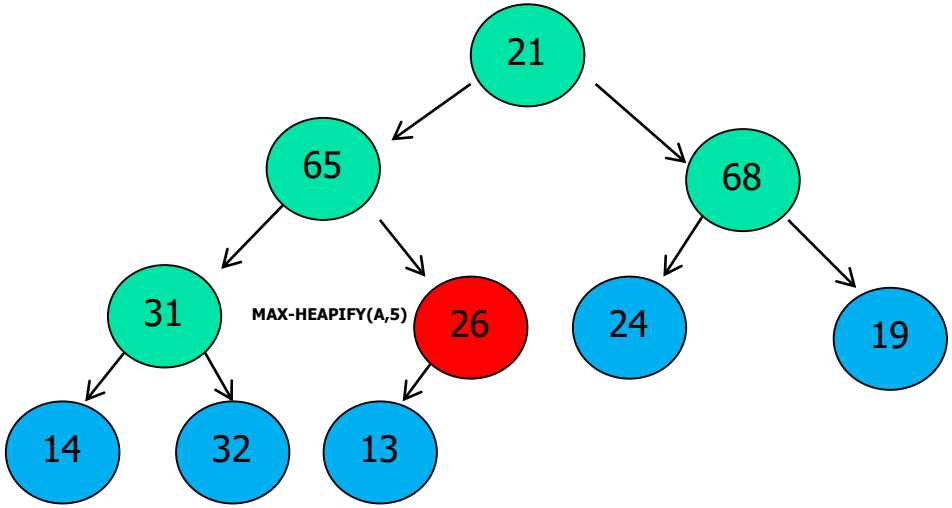
array		21	65	68	31	13	24	19	14	32	26	
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of BUILD-MAX-HEAP(A)



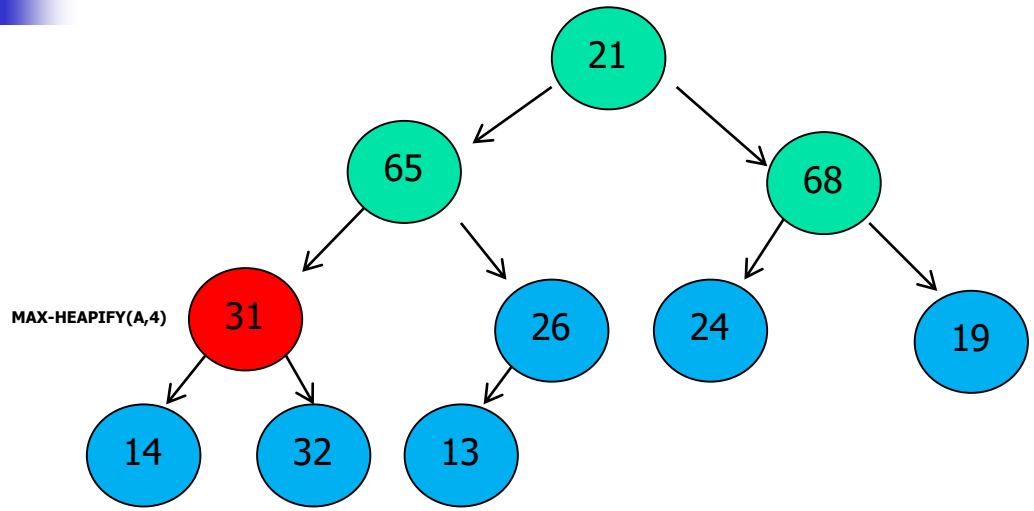
array		21	65	68	31	13	24	19	14	32	26	
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of BUILD-MAX-HEAP(A)



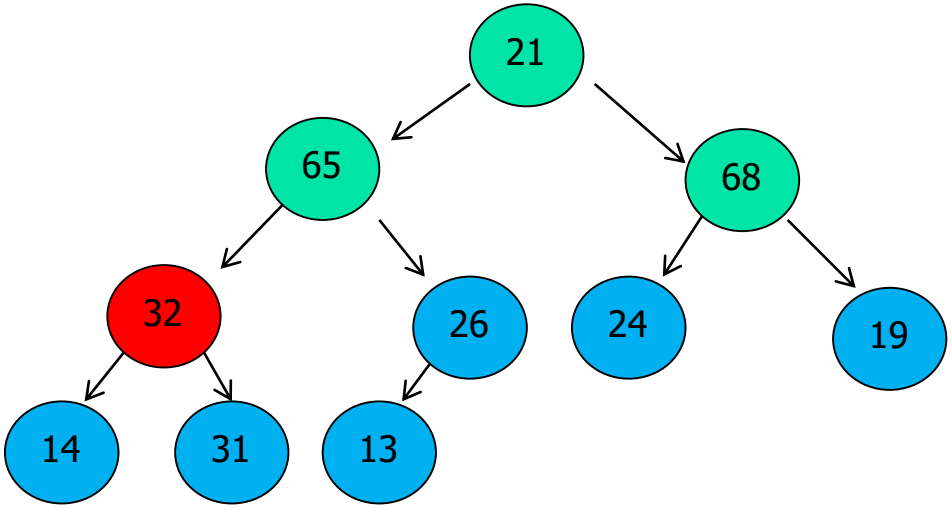
array		21	65	68	31	26	24	19	14	32	13		
index	0	1	2	3	4	5	6	7	8	9	10	11	

Operation of BUILD-MAX-HEAP(A)



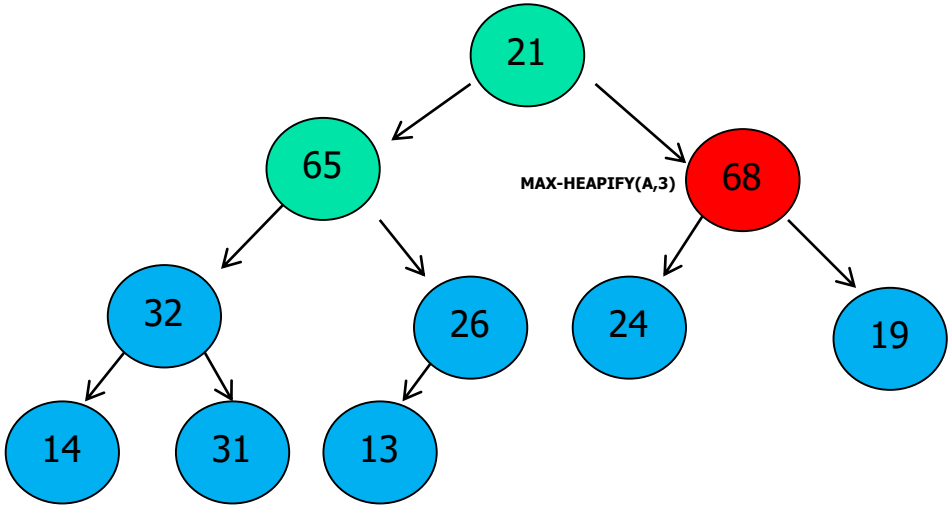
array		21	65	68	31	26	24	19	14	32	13	
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of BUILD-MAX-HEAP(A)



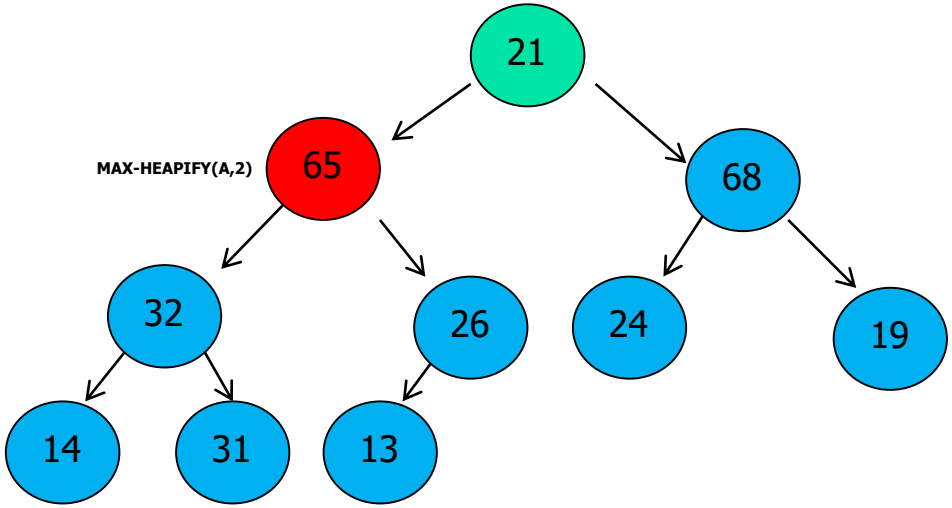
array		21	65	68	32	26	24	19	14	31	13	
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of BUILD-MAX-HEAP(A)



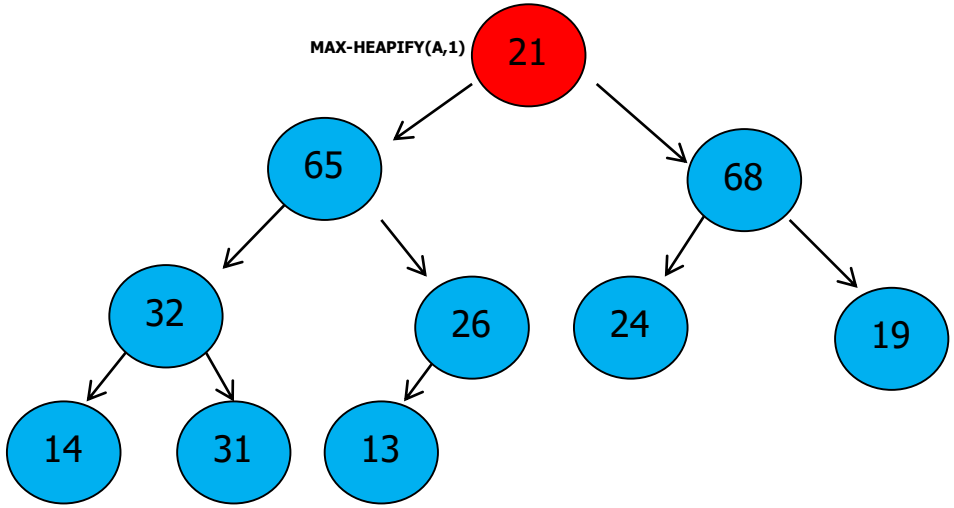
array		21	65	68	32	26	24	19	14	31	13	
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of BUILD-MAX-HEAP(A)



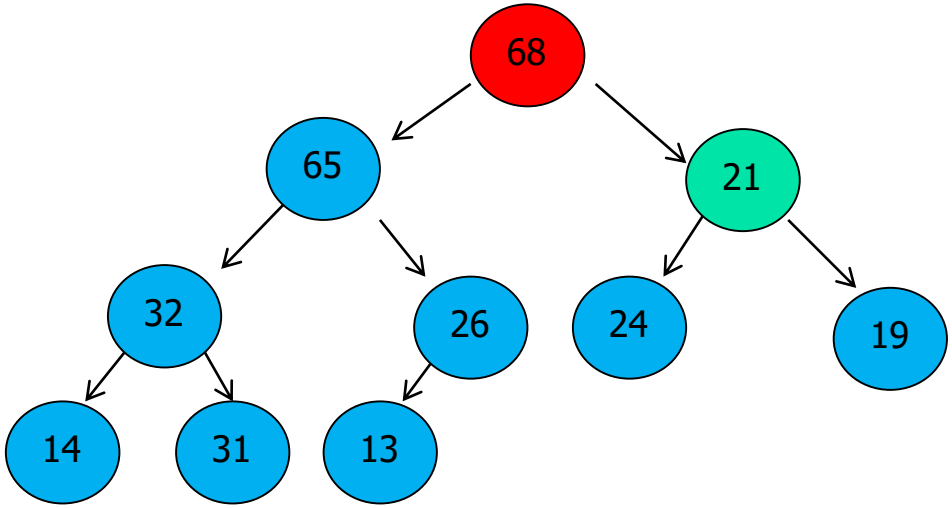
array		21	65	68	32	26	24	19	14	31	13		
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of BUILD-MAX-HEAP(A)



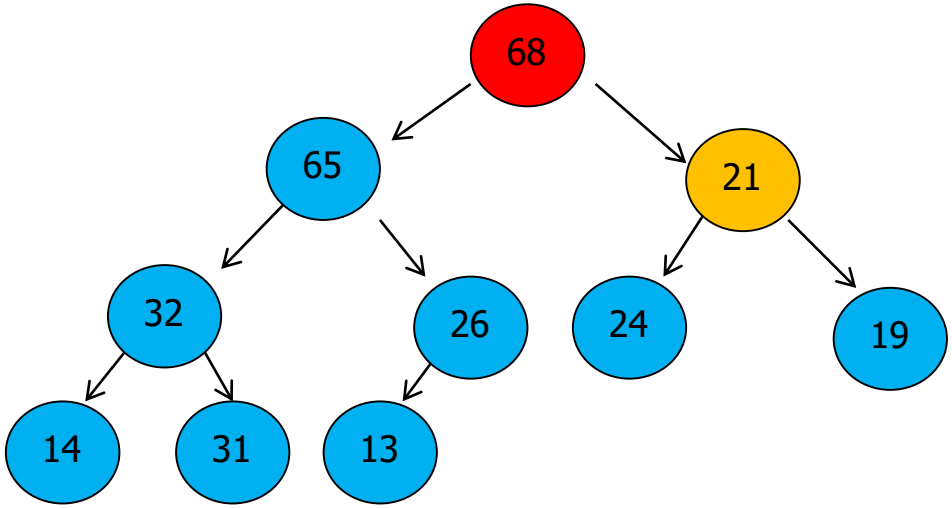
array		21	65	68	32	26	24	19	14	31	13	
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of BUILD-MAX-HEAP(A)



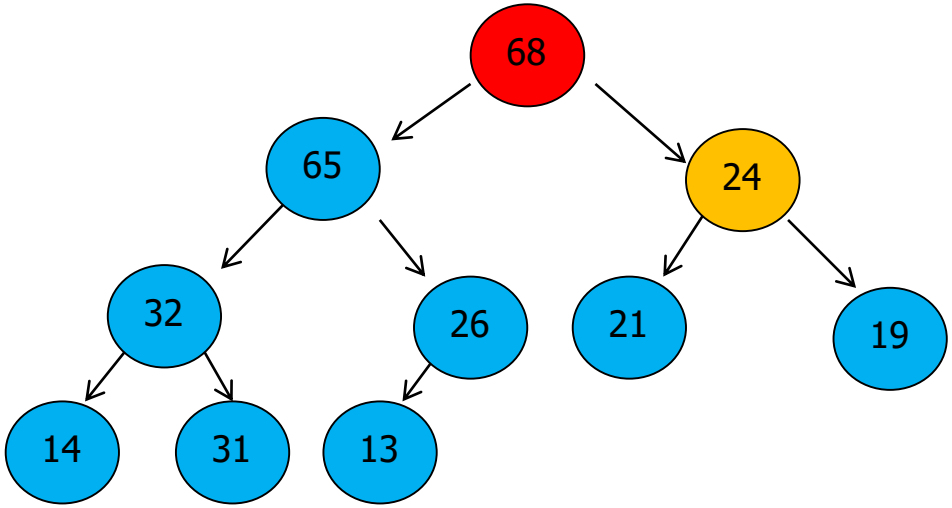
array		68	65	21	32	26	24	19	14	31	13	
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of BUILD-MAX-HEAP(A)



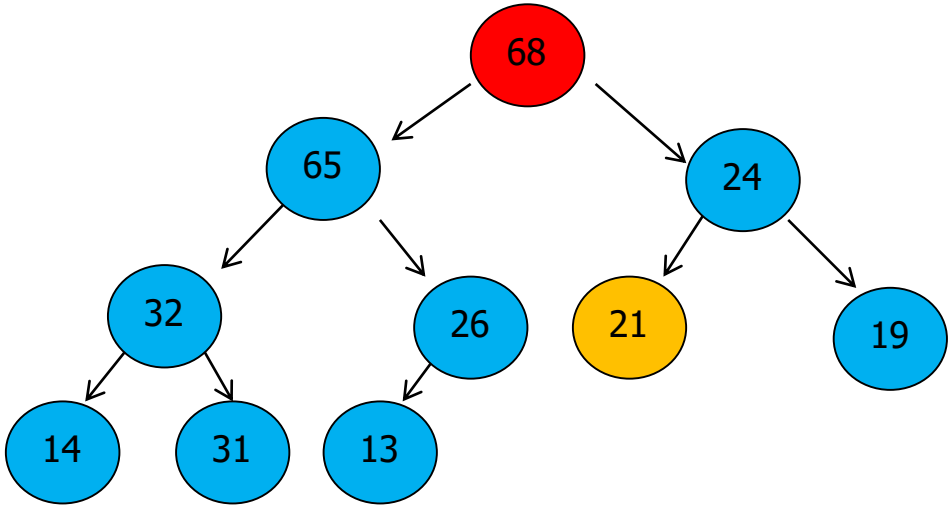
array		68	65	21	32	26	24	19	14	31	13	
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of BUILD-MAX-HEAP(A)



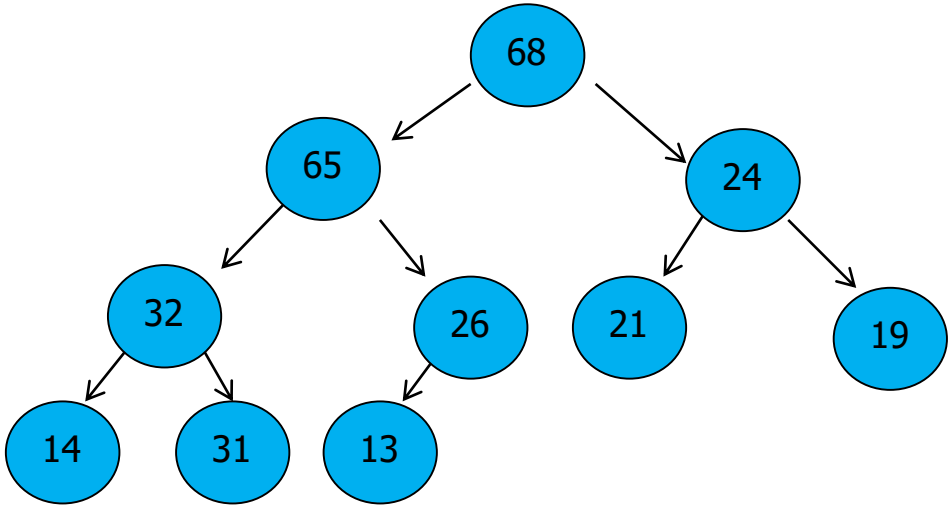
array		68	65	24	32	26	21	19	14	31	13	
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of BUILD-MAX-HEAP(A)



array		68	65	24	32	26	21	19	14	31	13	
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of BUILD-MAX-HEAP(A)



array		68	65	24	32	26	21	19	14	31	13	
index	0	1	2	3	4	5	6	7	8	9	10	11

BUILD-MAX-HEAP(A)

- We can use MAX-HEAPIFY in a bottom-up manner to convert array A[1..n], where n = A.length, into a max-heap

BUILD-MAX-HEAP(A)

1. A.heap-size = A.length
2. **for** i = $\lfloor A.length/2 \rfloor$ **downto** 1
3. MAX-HEAPIFY(A, i)

- Loop Invariant
  - At the start of each iteration of the for-loop of lines 2-3, each node i+1, i+2,…,n is the root of a max-heap.

Loop Invariants and the Correctness Proof

- We use loop invariants to help us understand why an algorithm is correct.
- We must show three things about a loop. invariant:
  - Initialization: It is true prior to the first iteration of the loop.
  - Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
  - Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Correctness Proof of BUILD-MAX-HEAP(A)

- Loop Invariant
  - At the start of each iteration of the for-loop of lines 2-3, each node i+1, i+2,…,n is the root of a max-heap.
- Initialization:
  - Prior to the first iteration of the loop, i = $\lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor$+1, $\lfloor n/2 \rfloor$+2,…,n is a leaf node which is the root of a trivial max-heap

Correctness Proof of BUILD-MAX-HEAP(A)

- Loop Invariant
  - At the start of each iteration of the for-loop of lines 2-3, each node i+1, i+2,…,n is the root of a max-heap.
- Maintenance:
  - By the loop invariant, both children of node i are the roots of max-heaps. Thus, MAX-HEAPIFY(A, i) makes the subtree rooted at node i a max-heap. It also preserves the property that the subtrees rooted at node i+1, i+2,…,n are max-heaps. Decreasing i in the for loop reestablishes the loop invariant for the next iteration.

Correctness Proof of BUILD-MAX-HEAP(A)

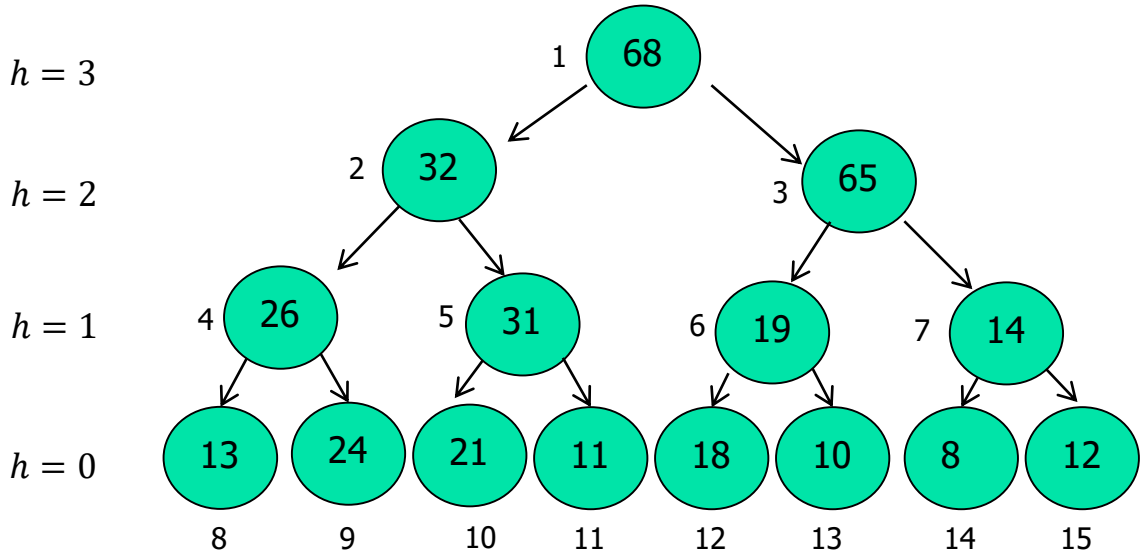
- Loop Invariant
  - At the start of each iteration of the for-loop of lines 2-3, each node i+1, i+2,…,n is the root of a max-heap.
- Termination:
  - At termination, i = 0. By the loop invariant, each node 1,2,…,n is the root of a max-heap. In other words, the subtree rooted at node 1 is a max-heap.

A Simple Upper Bound

- BUILD-MAX-HEAP takes O(n lg n) time.
  - BUILD-MAX-HEAP makes O(n) calls to MAX-HEAPIFY which costs O(lg n) time.

A Tighter Upper Bound

- MAX-HAEPIFY with a node of height h takes O(h) time.
- An n-element heap has height $\lfloor \lg n \rfloor$.
- Any heap with height h has at most $\lceil n/2^{h+1} \rceil$ nodes.



A Tighter Upper Bound

- MAX-HAEPIFY with a node of height h takes O(h) time.
- An n-element heap has height $\lfloor \lg n \rfloor$.
- Any heap with height h has at most $\lceil n/2^{h+1} \rceil$ nodes.
- Thus, BUILD-MAX-HEAP costs
  - $\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \leq \sum_{h=0}^{\lfloor \lg n \rfloor} \left(\frac{n}{2^{h+1}} + 1 \right) O(h)$

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

$$\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) + O((\lg n)^2)$$

$$= O(n) + O\left((\lg n)^2\right) = O(n)$$

- $S = \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{0}{2^0} + \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots$

$$\frac{1}{2} S = \frac{0}{2^1} + \frac{1}{2^2} + \frac{2}{2^3} + \cdots$$

$$S - \frac{1}{2} S = \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots = \frac{\frac{1}{2}}{1 - \frac{1}{2}} = 1$$

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

Another Analysis of Building a Heap

- Let's assume the tree is complete : $n = 2^h - 1$
  - There is one key at the height=h ,which might shift down h levels
  - There is two keys at the height=h-1 ,which might shift down h-1 levels
  - There is four key at the height=h-2 ,which might shift down h-2 levels
- Overall costs $S = h + 2(h-1) + 2^2(h-2) + \cdots + 2^{h-1}(1)$

$$2S = 2h + 2^2(h-1) + 2^3(h-2) + \cdots + 2^h(1)$$

$$-S = h - \left(2^1 + 2^2 + \cdots + 2^{h-1}\right) - 2^h$$

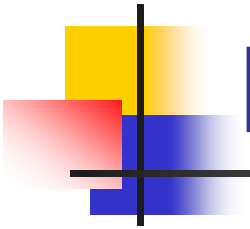
$$S = -h + (2^1 + 2^2 + \cdots + 2^h)$$

$$= -h - 1 + \left(2^0 + 2^1 + \cdots + 2^{h-1} + 2^h\right)$$

$$= 2^{h+1} - h - 2 = 2 \cdot 2^{\lg n} - \lg n - 2$$

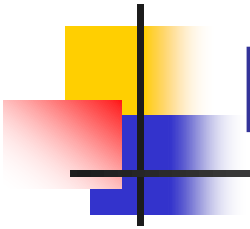
$$\leq 2n$$

$\alpha^{\lg \beta} = \beta^{\lg \alpha}$



Heapsort

- Use BUILD-MAX-HEAP to build a max-heap.
- Since the maximum element of the array is stored at the root A[1], put it into its correct position by exchanging it with A[n].
- Restore the max-heap property by calling MAX-HEAPIFY(A, 1) to generate a max-heap in A[1..n-1].
- Repeat the steps for the max-heap of size n-1 down to a heap of size 2

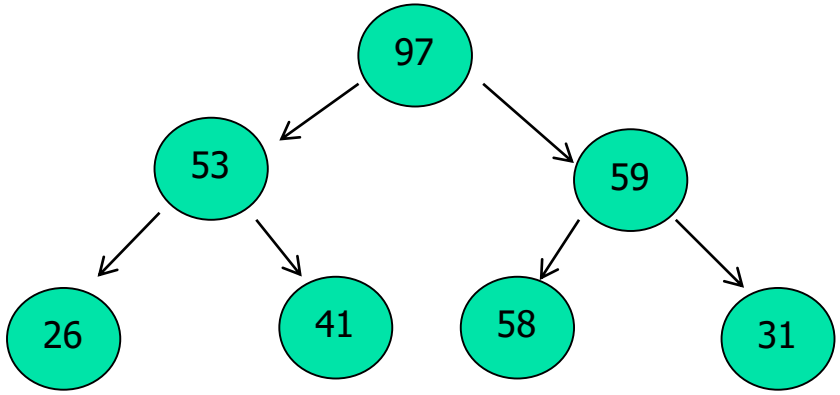


Heapsort(A)

HAEPSORT(A)

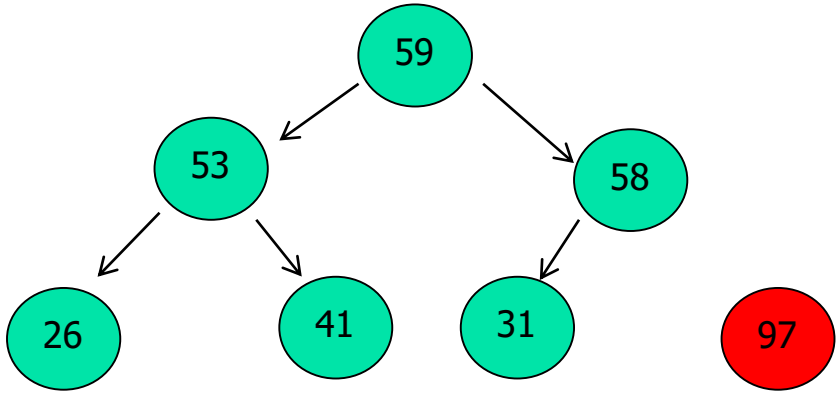
1. BUILD-MAX-HEAP(A)
2. **for** i = A.length **downto** 2
3. exchange A[1] with A[i]
4. A.heap-size = A.heap-size-1
5. MAX-HEAPIFY(A,1)

Operation of Heap Sort



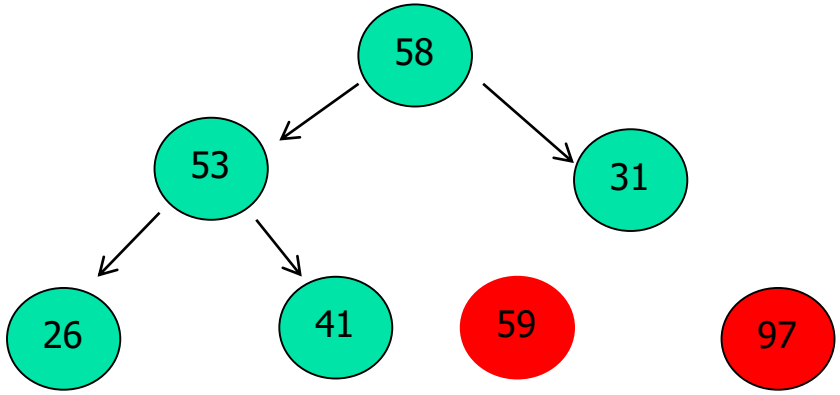
array		97	53	59	26	41	58	31				
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of Heap Sort



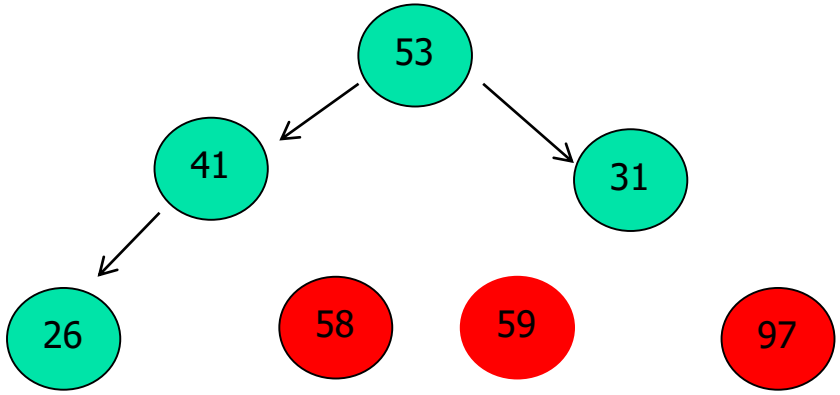
array		59	53	58	26	41	31	97				
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of Heap Sort



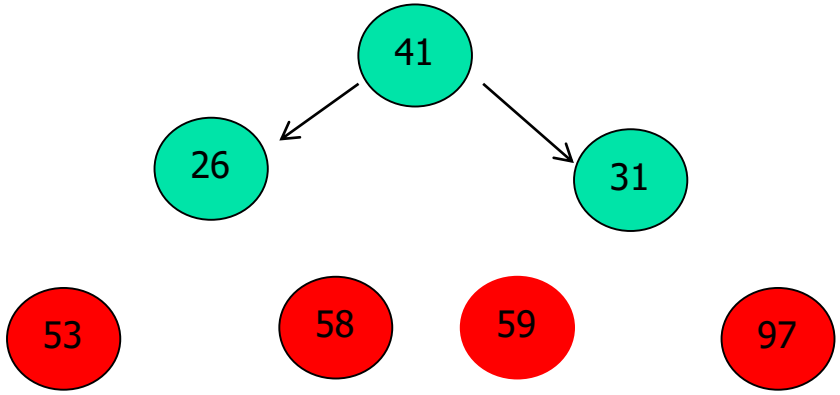
array		58	53	31	26	41	59	97				
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of Heap Sort



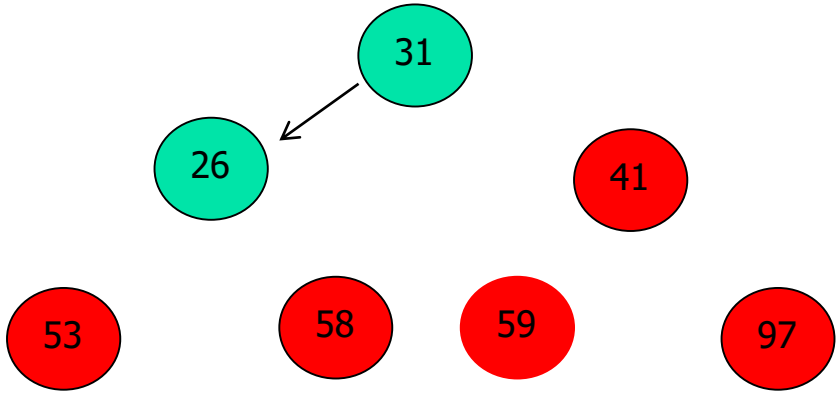
array		53	41	31	26	58	59	97				
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of Heap Sort



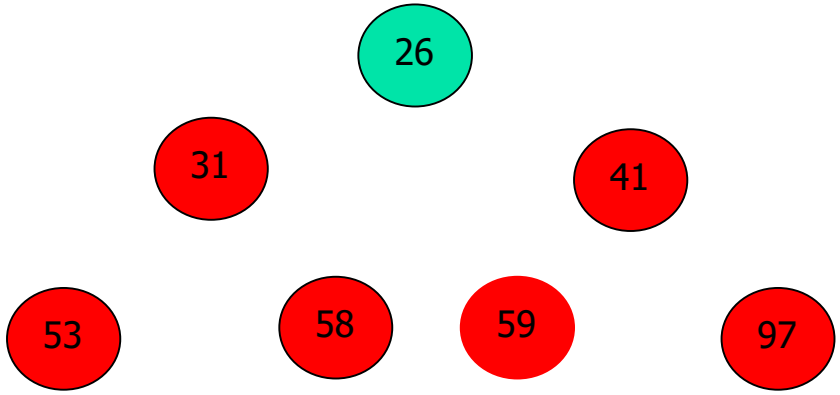
array		41	26	31	53	58	59	97				
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of Heap Sort

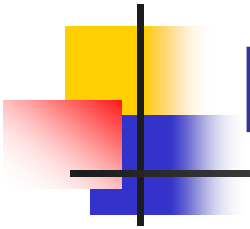


array		31	26	41	53	58	59	97				
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of Heap Sort

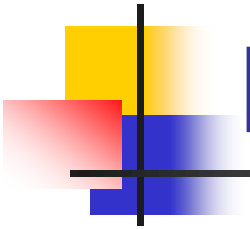


array		26	31	41	53	58	59	97				
index	0	1	2	3	4	5	6	7	8	9	10	11



Heapsort

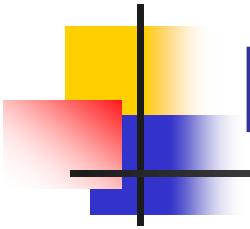
- It takes O(n lg n) time:
  - Calls to BUILD-MAX-HEAP takes O(n) time.
  - Each of the n-1 calls to MAX-HEAPIFY takes O(lg n) time.



Priority Queues

- HEAP-MAXIMUM(A)

1. Return A[1]



Priority Queues

- HEAP-EXTRACT-MAX(A)

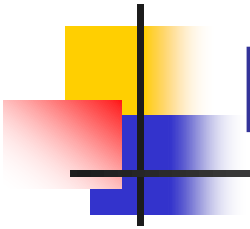
```
1.  if A.heap-size < 1
2.      error "heap underflow"
3.  max = A[1]
4.  A[1] = A[A.heap-size]
5.  A.heap-size = A.heap-size – 1
6.  MAX-HEAPIFY(A, 1)
7.  return max
```

Priority Queues

- HEAP-INCREASE-KEY(A, i , key)

```
1.   if key < A[i]
2.      error "new key is smaller than current key"
3.   A[i] = key
4.   while i > 1 and A[PARENT(i)] < A[i]
5.      exchange A[i] with A[PARENT(i)]
6.      i = PARENT(i)
```

- An index i into the array identifies the priority-queue element whose key we wish to increase
- The procedure first updates the key of element A[i] to its new value
- Because increasing the key of A[i] might violate the max-heap property, it traverses a simple path from this node toward to the root to find a proper place for the newly increased key



Priority Queues

- MAX-HEAP-INSERT(A, key)

1. A.heap-size = A.heap-size + 1
2. A[A.heap-size] = -∞
3. HEAP-INCREASE-KEY(A, A.heap-size, key)

Any Question?