Lecture 8
Heaps, Heapsort, Optimality of Heapsort/Mergesort (revisited)

CS 161 Design and Analysis of Algorithms

Ioannis Panageas

Heapsort

Consider the following version of Selection Sort (sometimes called Max sort)

```
def maxSort(A,n):
    for k = n-1 downto 1
        find j such that A[j] == max(A[0],A[1],..., A[k])
        A[j] ↔ A[k]
```

Heapsort

Consider the following version of Selection Sort (sometimes called Max sort)

```
def maxSort(A,n):
    for k = n-1 downto 1
        find j such that A[j] == max(A[0],A[1],..., A[k])
        A[j] ↔ A[k]
```

A straightforward implementation requires $O(n^2)$ time, because of the time spent repeatedly finding the maximum of the first k items.

Heapsort

Consider the following version of Selection Sort (sometimes called Max sort)

```
def maxSort(A,n):
    for k = n-1 downto 1
        find j such that A[j] == max(A[0],A[1],..., A[k])
        A[j] ↔ A[k]
```

A straightforward implementation requires $O(n^2)$ time, because of the time spent repeatedly finding the maximum of the first k items.

But we can speed this up by using a binary heap.

Priority Queues and Heaps

- Priority Queue

Priority Queues and Heaps

- Priority Queue
  - Abstract data type

Priority Queues and Heaps

- Priority Queue
  - Abstract data type
  - Collection of items.

Priority Queues and Heaps

- Priority Queue
  - Abstract data type
  - Collection of items.
  - Each item has an associated key, which corresponds to a priority.

Priority Queues and Heaps

- Priority Queue
  - Abstract data type
  - Collection of items.
  - Each item has an associated key, which corresponds to a priority.
  - Supports the following operations

Priority Queues and Heaps

- Priority Queue
  - Abstract data type
  - Collection of items.
  - Each item has an associated key, which corresponds to a priority.
  - Supports the following operations
    - Insert an item with a given key

Priority Queues and Heaps

- Priority Queue
  - Abstract data type
  - Collection of items.
  - Each item has an associated key, which corresponds to a priority.
  - Supports the following operations
    - Insert an item with a given key
    - Delete an item

Priority Queues and Heaps

- Priority Queue
  - Abstract data type
  - Collection of items.
  - Each item has an associated key, which corresponds to a priority.
  - Supports the following operations
    - Insert an item with a given key
    - Delete an item
    - Select the item with the most urgent priority in the priority queue.

Priority Queues and Heaps

- Priority Queue
  - Abstract data type
  - Collection of items.
  - Each item has an associated key, which corresponds to a priority.
  - Supports the following operations
    - Insert an item with a given key
    - Delete an item
    - Select the item with the most urgent priority in the priority queue.
  - Most urgent priority may correspond to the lowest key value or to the highest key value, depending on the application.

Binary Heaps

- Specific implementation of priority queue

Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.

Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.
- The array represents a binary tree in level order (breadth-first order).

Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.
- The array represents a binary tree in level order (breadth-first order).
- Can be max-heap or min-heap

Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.
- The array represents a binary tree in level order (breadth-first order).
- Can be max-heap or min-heap
  - In a max-heap, large key values represent more urgent priorities

Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.
- The array represents a binary tree in level order (breadth-first order).
- Can be max-heap or min-heap
  - In a max-heap, large key values represent more urgent priorities
  - In a min-heap, small key values represent more urgent priorities

Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.
- The array represents a binary tree in level order (breadth-first order).
- Can be max-heap or min-heap
  - In a max-heap, large key values represent more urgent priorities
  - In a min-heap, small key values represent more urgent priorities
- In this introduction, we will be using a max-heap.

Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.
- The array represents a binary tree in level order (breadth-first order).
- Can be max-heap or min-heap
  - In a max-heap, large key values represent more urgent priorities
  - In a min-heap, small key values represent more urgent priorities
- In this introduction, we will be using a max-heap.
- Heap invariant for max-heaps: For any item v other than the root,

$$\texttt{key}\,(\texttt{parent}(v)) \geq \texttt{key}(v)$$

Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.
- The array represents a binary tree in level order (breadth-first order).
- Can be max-heap or min-heap
  - In a max-heap, large key values represent more urgent priorities
  - In a min-heap, small key values represent more urgent priorities
- In this introduction, we will be using a max-heap.
- Heap invariant for max-heaps: For any item v other than the root,

$$\texttt{key}\,(\texttt{parent}(v)) \geq \texttt{key}(v)$$

- In a min-heap, the direction of the inequality is reversed.

Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.
- The array represents a binary tree in level order (breadth-first order).
- Can be max-heap or min-heap
  - In a max-heap, large key values represent more urgent priorities
  - In a min-heap, small key values represent more urgent priorities
- In this introduction, we will be using a max-heap.
- Heap invariant for max-heaps: For any item v other than the root,

$$\texttt{key}\,(\texttt{parent}(v)) \geq \texttt{key}(v)$$

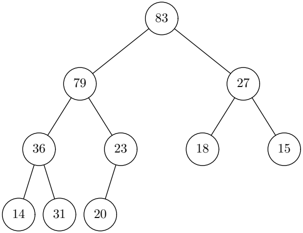
- In a min-heap, the direction of the inequality is reversed.
- In our examples, items are integers, key is the integer value

Viewing the array as a binary tree

83	79	27	36	23	18	15	14	31	20
0	1	2	3	4	5	6	7	8	9

Viewing the array as a binary tree

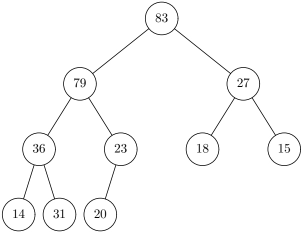
83	79	27	36	23	18	15	14	31	20
0	1	2	3	4	5	6	7	8	9



Viewing the array as a binary tree

- Root is $H[0]$

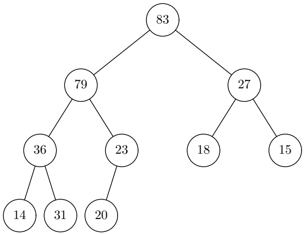
83	79	27	36	23	18	15	14	31	20
0	1	2	3	4	5	6	7	8	9



Viewing the array as a binary tree

- Root is $H[0]$
- Left child of $H[i]$ is $H[2i+1]$ (provided $2i+1 < n$, where $n = H.\text{size}$)

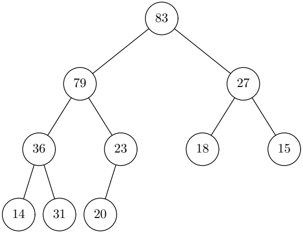
83	79	27	36	23	18	15	14	31	20
0	1	2	3	4	5	6	7	8	9



Viewing the array as a binary tree

- Root is $H[0]$
- Left child of $H[i]$ is $H[2i+1]$ (provided $2i+1<n$, where $n=H.\text{size}$)
- Right child of $H[i]$ is $H[2i+2]$ (provided $2i+2<n$)

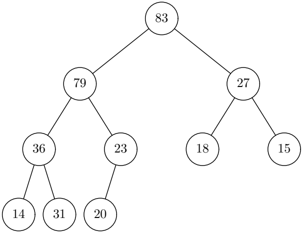
83	79	27	36	23	18	15	14	31	20
0	1	2	3	4	5	6	7	8	9



Viewing the array as a binary tree

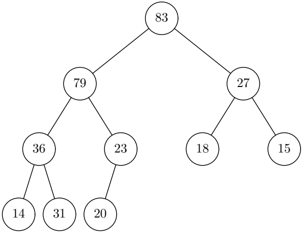
- Root is $H[0]$
- Left child of $H[i]$ is $H[2i+1]$ (provided $2i+1 < n$, where $n = H.\text{size}$)
- Right child of $H[i]$ is $H[2i+2]$ (provided $2i+2 < n$)
- Parent of $H[i]$ is $H[\lfloor (i-1)/2 \rfloor]$ (provided $i > 0$)

83	79	27	36	23	18	15	14	31	20
0	1	2	3	4	5	6	7	8	9



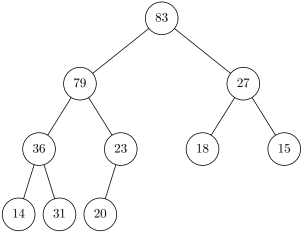
Heap operations in a max-heap:

Heap operations in a max-heap:



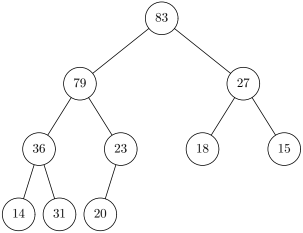
Heap operations in a max-heap:

- `FindMax(H)`: Find maximum item in the heap



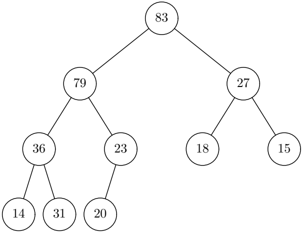
Heap operations in a max-heap:

- `FindMax(H)`: Find maximum item in the heap
- `ExtractMax(H)`: Find maximum item and delete it from the heap



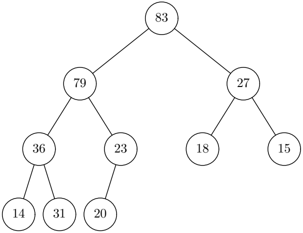
Heap operations in a max-heap:

- `FindMax(H)`: Find maximum item in the heap
- `ExtractMax(H)`: Find maximum item and delete it from the heap
- `Insert(H,x)`: Insert the new item x in the heap



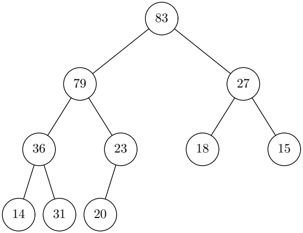
Heap operations in a max-heap:

- `FindMax(H)`: Find maximum item in the heap
- `ExtractMax(H)`: Find maximum item and delete it from the heap
- `Insert(H,x)`: Insert the new item x in the heap
- `Delete(H,i)`: Delete the item at location i from the heap



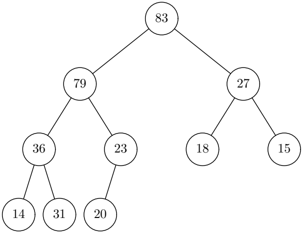
`FindMax`: Find maximum item in the heap

FindMax: Find maximum item in the heap



FindMax: Find maximum item in the heap

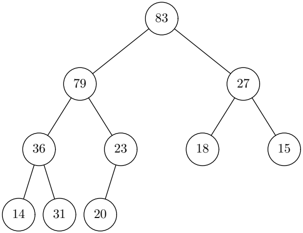
Findmax is easy: just report the value at the root.



FindMax: Find maximum item in the heap

Findmax is easy: just report the value at the root.

```
def FindMax(H):
    return H[0]
```



Helper functions

Helper functions

- Except for FindMax, the binary heap operations require some data movement.

Helper functions

- Except for FindMax, the binary heap operations require some data movement.
- The heap invariant must be preserved after each operation.

Helper functions

- Except for FindMax, the binary heap operations require some data movement.
- The heap invariant must be preserved after each operation.
- We define two helper functions.

Helper functions

- Except for FindMax, the binary heap operations require some data movement.
- The heap invariant must be preserved after each operation.
- We define two helper functions.
  - `SiftUp(H,i)`: Move the item at location i up to its correct position by repeatedly swapping the item with its parent, as necessary.

Helper functions

- Except for FindMax, the binary heap operations require some data movement.
- The heap invariant must be preserved after each operation.
- We define two helper functions.
  - `SiftUp(H,i)`: Move the item at location i up to its correct position by repeatedly swapping the item with its parent, as necessary.
  - `SiftDown(H,i)`: Move the item at location i down to its correct position by repeatedly swapping the item with the child having the larger key, as necessary.

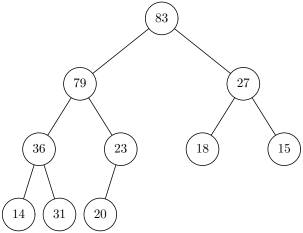
Helper functions

- Except for FindMax, the binary heap operations require some data movement.
- The heap invariant must be preserved after each operation.
- We define two helper functions.
  - `SiftUp(H,i)`: Move the item at location *i* up to its correct position by repeatedly swapping the item with its parent, as necessary.
  - `SiftDown(H,i)`: Move the item at location *i* down to its correct position by repeatedly swapping the item with the child having the larger key, as necessary.

[GT] calls these "up-heap bubbling" and "down-heap bubbling"

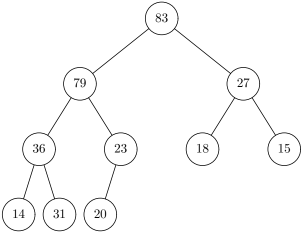
`SiftUp`: Sift an item up to its correct position

SiftUp: Sift an item up to its correct position



SiftUp: Sift an item up to its correct position

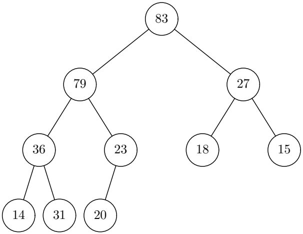
```
def SiftUp(H,i):
    parent = (i-1)/2;
    if (i > 0) and (H[parent].key < H[i].key):
        H[i] ↔ H[parent]
        SiftUp(H,parent)
```



SiftUp: Sift an item up to its correct position

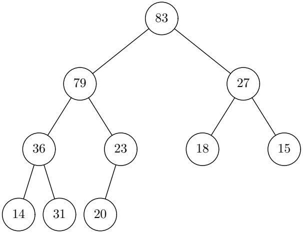
```
def SiftUp(H,i):
    parent = (i-1)/2;
    if (i > 0) and (H[parent].key < H[i].key):
        H[i] ↔ H[parent]
        SiftUp(H,parent)
```

Analysis: at most 1 comparison at each level, so total time is $O(\log n)$



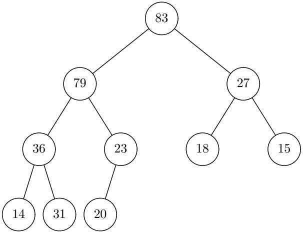
`SiftDown`: Sift an item down to its correct position

SiftDown: Sift an item down to its correct position



SiftDown: Sift an item down to its correct position

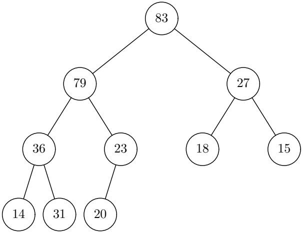
```
def SiftDown(H,i):
    n = H.size // number of item in heap
    left = 2i+1; right = 2i+2
    if (right < n) and (H[right].key > H[left].key)
        largerChild = right
    else largerChild = left
    if (largerchild < n) and (H[i].key < H[largerChild].key)
        H[i] ↔ H[largerchild]
        SiftDown(H,largerchild)
```



SiftDown: Sift an item down to its correct position

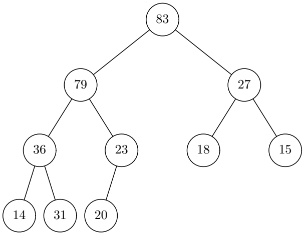
```
def SiftDown(H,i):
    n = H.size // number of item in heap
    left = 2i+1; right = 2i+2
    if (right < n) and (H[right].key > H[left].key)
        largerChild = right
    else largerChild = left
    if (largerchild < n) and (H[i].key < H[largerChild].key)
        H[i] ↔ H[largerchild]
        SiftDown(H,largerchild)
```

Analysis: at most 2 comparisons at each level, so total time is $O(\log n)$



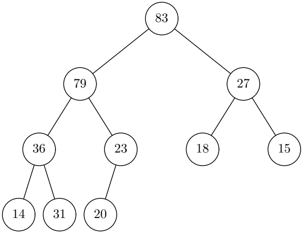
Insert: Insert the new item x

Insert: Insert the new item x



Insert: Insert the new item x

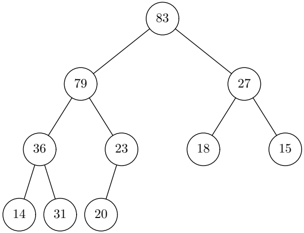
```
def Insert(H,x):
    H.size = H.size+1 // increment number of items
    k = H.size-1 //index of last position
    H[k] = x //insert x in last position
    SiftUp(H,k)
```



Insert: Insert the new item x

```
def Insert(H,x):
    H.size = H.size+1 // increment number of items
    k = H.size-1 //index of last position
    H[k] = x //insert x in last position
    SiftUp(H,k)
```

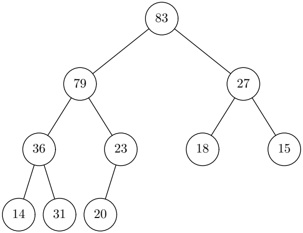
Analysis: Siftup time dominates, so total time is $O(\log n)$



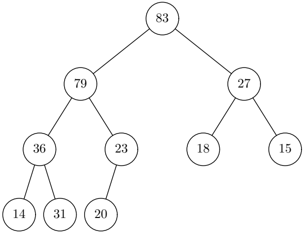
Insert: Insert the new item x

```
def Insert(H,x):
    H.size = H.size+1 // increment number of items
    k = H.size-1 //index of last position
    H[k] = x //insert x in last position
    SiftUp(H,k)
```

Analysis: Siftup time dominates, so total time is $O(\log n)$



`Insert(H,81)`

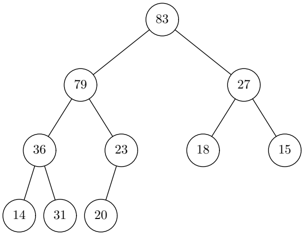


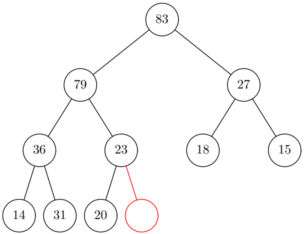
Insert: Insert the new item x

```
def Insert(H,x):
    H.size = H.size+1 // increment number of items
    k = H.size-1 //index of last position
    H[k] = x //insert x in last position
    SiftUp(H,k)
```

Analysis: Siftup time dominates, so total time is $O(\log n)$

Insert(H,81)

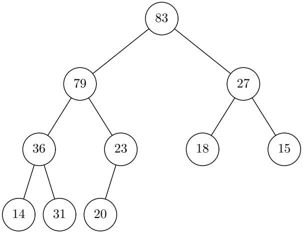




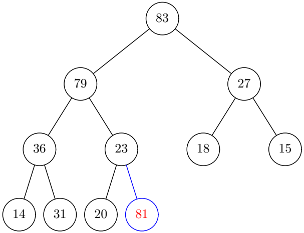
Insert: Insert the new item x

```
def Insert(H,x):
    H.size = H.size+1 // increment number of items
    k = H.size-1 //index of last position
    H[k] = x //insert x in last position
    SiftUp(H,k)
```

Analysis: Siftup time dominates, so total time is $O(\log n)$



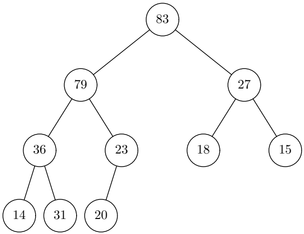
`Insert(H,81)`



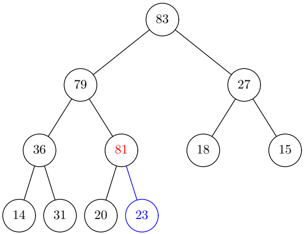
Insert: Insert the new item x

```
def Insert(H,x):
    H.size = H.size+1 // increment number of items
    k = H.size-1 //index of last position
    H[k] = x //insert x in last position
    SiftUp(H,k)
```

Analysis: Siftup time dominates, so total time is $O(\log n)$



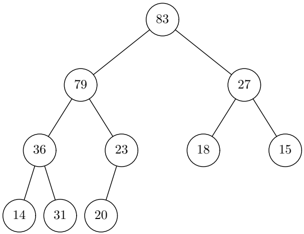
`Insert(H,81)`



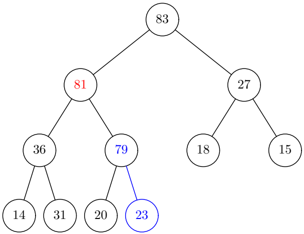
Insert: Insert the new item x

```
def Insert(H,x):
    H.size = H.size+1 // increment number of items
    k = H.size-1 //index of last position
    H[k] = x //insert x in last position
    SiftUp(H,k)
```

Analysis: Siftup time dominates, so total time is $O(\log n)$



Insert(H,81)

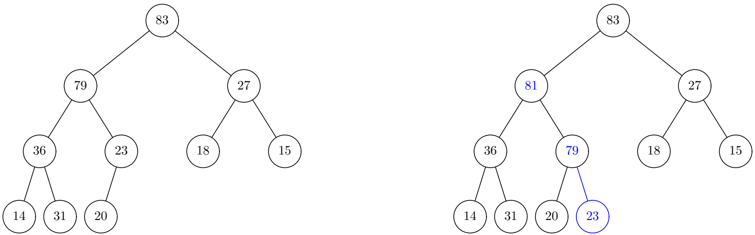


Insert: Insert the new item x

```
def Insert(H,x):
    H.size = H.size+1 // increment number of items
    k = H.size-1 //index of last position
    H[k] = x //insert x in last position
    SiftUp(H,k)
```

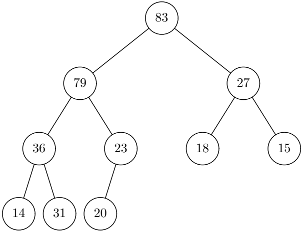
Analysis: Siftup time dominates, so total time is $O(\log n)$

Insert(H,81)



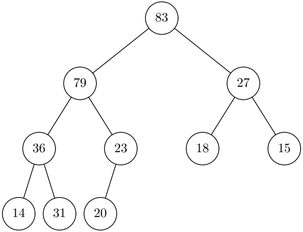
Delete: Delete the item at location i

Delete: Delete the item at location i



Delete: Delete the item at location *i*

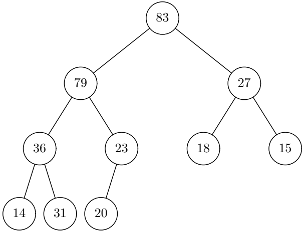
```
def Delete(H,i):
    k = H.size-1 //index of last position
    H[i] = H[k] // overwrite item being deleted with
                element in last position
    H.size = H.size-1 // decrement number of item
    SiftUp(H,i) // either SiftUp or SiftDown will do nothing
    SiftDown(H,i)
```



Delete: Delete the item at location i

```
def Delete(H,i):
    k = H.size-1 //index of last position
    H[i] = H[k] // overwrite item being deleted with
                   element in last position
    H.size = H.size-1 // decrement number of item
    SiftUp(H,i) // either SiftUp or SiftDown will do nothing
    SiftDown(H,i)
```

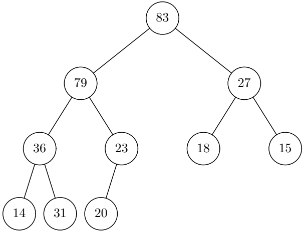
Analysis: Siftup/siftdown time dominates, so total time is $O(\log n)$



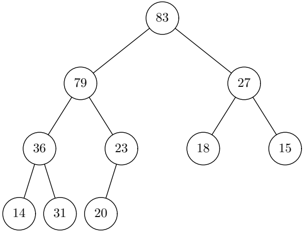
Delete: Delete the item at location i

```
def Delete(H,i):
    k = H.size-1 //index of last position
    H[i] = H[k] // overwrite item being deleted with
                element in last position
    H.size = H.size-1 // decrement number of item
    SiftUp(H,i) // either SiftUp or SiftDown will do nothing
    SiftDown(H,i)
```

Analysis: Siftup/siftdown time dominates, so total time is $O(\log n)$



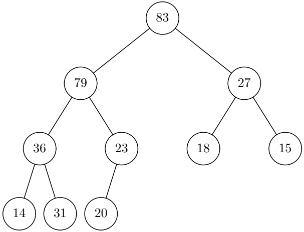
Delete(H,3)



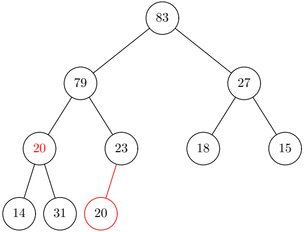
Delete: Delete the item at location i

```
def Delete(H,i):
    k = H.size-1 //index of last position
    H[i] = H[k] // overwrite item being deleted with
                   element in last position
    H.size = H.size-1 // decrement number of item
    SiftUp(H,i) // either SiftUp or SiftDown will do nothing
    SiftDown(H,i)
```

Analysis: Siftup/siftdown time dominates, so total time is $O(\log n)$



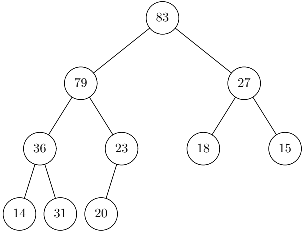
Delete(H,3)



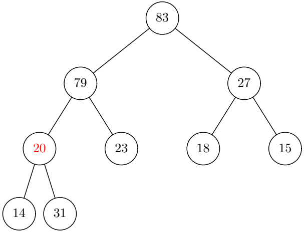
Delete: Delete the item at location i

```
def Delete(H,i):
    k = H.size-1 //index of last position
    H[i] = H[k] // overwrite item being deleted with
                   element in last position
    H.size = H.size-1 // decrement number of item
    SiftUp(H,i) // either SiftUp or SiftDown will do nothing
    SiftDown(H,i)
```

Analysis: Siftup/siftdown time dominates, so total time is $O(\log n)$



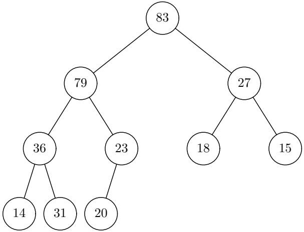
Delete(H,3)



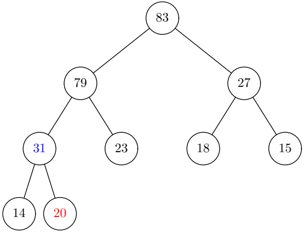
Delete: Delete the item at location i

```
def Delete(H,i):
    k = H.size-1 //index of last position
    H[i] = H[k] // overwrite item being deleted with
                   element in last position
    H.size = H.size-1 // decrement number of item
    SiftUp(H,i) // either SiftUp or SiftDown will do nothing
    SiftDown(H,i)
```

Analysis: Siftup/siftdown time dominates, so total time is $O(\log n)$



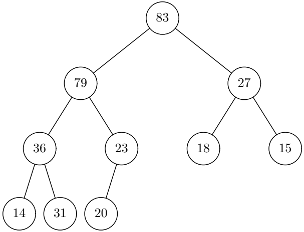
Delete(H,3)



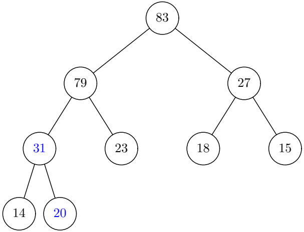
Delete: Delete the item at location *i*

```
def Delete(H,i):
    k = H.size-1 //index of last position
    H[i] = H[k] // overwrite item being deleted with
                   element in last position
    H.size = H.size-1 // decrement number of item
    SiftUp(H,i) // either SiftUp or SiftDown will do nothing
    SiftDown(H,i)
```

Analysis: Siftup/siftdown time dominates, so total time is $O(\log n)$

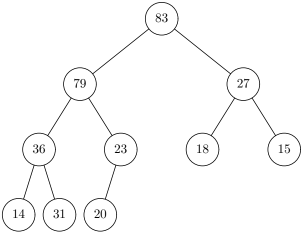


Delete(H,3)



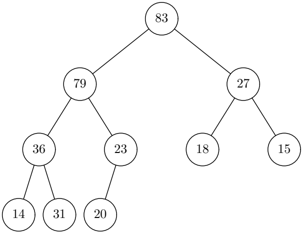
ExtractMax: Find maximum item and delete it

ExtractMax: Find maximum item and delete it



ExtractMax: Find maximum item and delete it

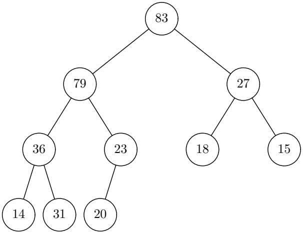
```
def ExtractMax(H):
    x = H[0]
    Delete(H,0)
    return x
```



ExtractMax: Find maximum item and delete it

```
def ExtractMax(H):
    x = H[0]
    Delete(H,0)
    return x
```

Analysis: Delete time dominates, so total time is $O(\log n)$



Constructing a heap

Constructing a heap

How do we efficiently construct a brand-new heap storing n given item?

Constructing a heap

How do we efficiently construct a brand-new heap storing n given item?

If we insert the items one at a time, time spent on kth insertion is $O(\log k)$.

Constructing a heap

How do we efficiently construct a brand-new heap storing n given item?

If we insert the items one at a time, time spent on kth insertion is $O(\log k)$.

So total time is

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

Constructing a heap

How do we efficiently construct a brand-new heap storing n given item?

If we insert the items one at a time, time spent on kth insertion is $O(\log k)$.

So total time is

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

There is a better way that only requires $O(n)$ time...

Constructing a heap in $O(n)$ time

Constructing a heap in $O(n)$ time

1. Put the data in H, in arbitrary order. (So H stores the correct data, but does not satisfy the heap invariant.)

Constructing a heap in $O(n)$ time

1. Put the data in H, in arbitrary order. (So H stores the correct data, but does not satisfy the heap invariant.)
2. Run the following `Heapify` function.

```
def heapify(H,n)

    for i = n-1 down to 0:
        SiftDown(H,i)
```

Constructing a heap in $O(n)$ time

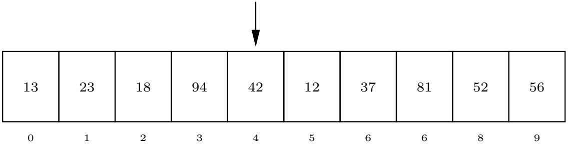
1. Put the data in H, in arbitrary order. (So H stores the correct data, but does not satisfy the heap invariant.)
2. Run the following `Heapify` function.

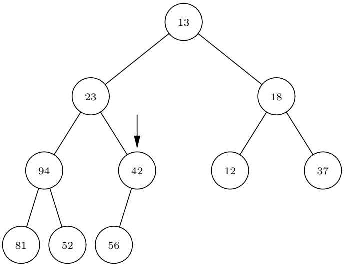
```
def heapify(H,n)
    for i = n-1 down to 0:
        SiftDown(H,i)
```

The code given above can be improved: We can start at $\mathrm{i} = \lfloor (\mathrm{n}\text{-}2)/2 \rfloor$ (or equivalently, $\mathrm{i} = \lfloor \mathrm{n}/2 \rfloor - 1$), rather than $i = n - 1$.

Heapify example

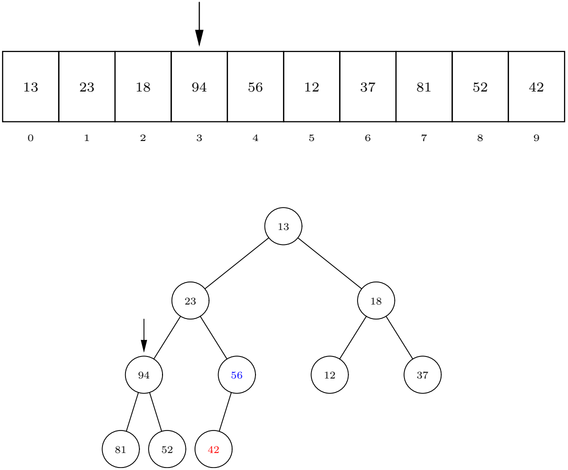
```
13 23 18 94 42 12 37 81 52 56
```





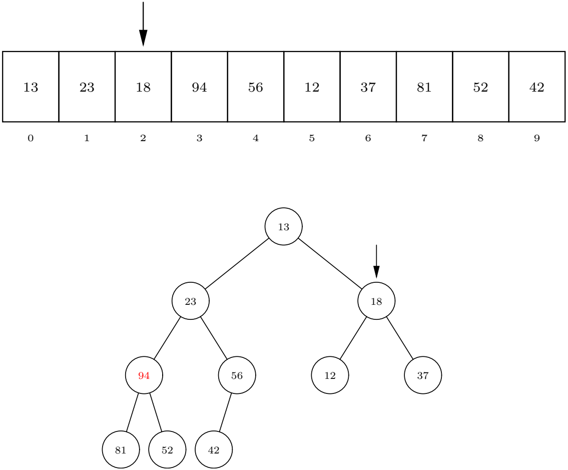
Heapify example, continued

13 23 18 94 42 12 37 81 52 56



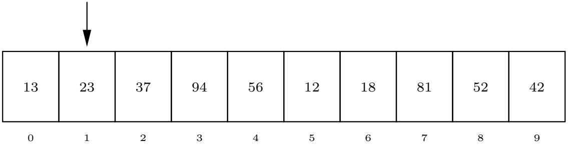
Heapify example, continued

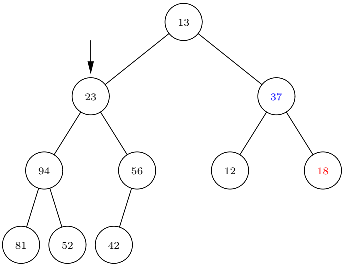
13 23 18 94 42 12 37 81 52 56



Heapify example, continued

13 23 18 94 42 12 37 81 52 56

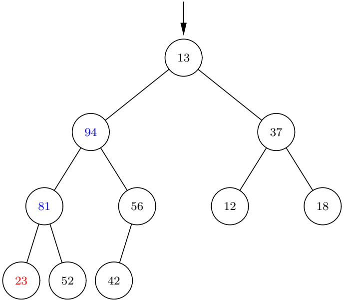




Heapify example, continued

13 23 18 94 42 12 37 81 52 56

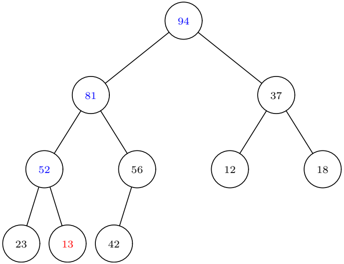
13	94	37	81	56	12	18	23	52	42
0	1	2	3	4	5	6	7	8	9



Heapify example, continued

13 23 18 94 42 12 37 81 52 56

94	81	37	52	56	12	18	23	13	42
0	1	2	3	4	5	6	7	9	9



Analysis of heap construction algorithm using Heapify

Analysis of heap construction algorithm using Heapify

```
Algorithm heapify(H,n);
   for i = n-1 down to 0:
      SiftDown(H,i)
```

Analysis of heap construction algorithm using Heapify

```
Algorithm heapify(H,n);
   for i = n-1 down to 0:
      SiftDown(H,i)
```

- Correctness: After SiftDown(H,i) is executed, subtree rooted at node *i* satisfies heap invariant. (Can show by induction).

Analysis of heap construction algorithm using Heapify

```
Algorithm heapify(H,n);
   for i = n-1 down to 0:
      SiftDown(H,i)
```

- Correctness: After SiftDown(H,i) is executed, subtree rooted at node i satisfies heap invariant. (Can show by induction).
- Running time: `Heapify` runs in $O(n)$ time. We will prove this on the next slide.

Proof that Heapify runs in $O(n)$ time

Proof that Heapify runs in $O(n)$ time

- Suppose the tree has n nodes and d levels (so $2^d \leq n < 2^{d+1}$).

Proof that Heapify runs in $O(n)$ time

- Suppose the tree has n nodes and d levels (so $2^d \leq n < 2^{d+1}$).
- If node i is at level j, `SiftDown(H,i)` needs $\leq 2(d-j)$ comparisons.

Proof that Heapify runs in $O(n)$ time

- Suppose the tree has n nodes and d levels (so $2^d \leq n < 2^{d+1}$).
- If node i is at level j, `SiftDown(H,i)` needs $\leq 2(d-j)$ comparisons.
- There are at most 2^j nodes at level j.

Proof that Heapify runs in $O(n)$ time

- Suppose the tree has n nodes and d levels (so $2^d \leq n < 2^{d+1}$).
- If node i is at level j, `SiftDown(H,i)` needs $\leq 2(d-j)$ comparisons.
- There are at most 2^j nodes at level j.
- So total number of comparisons is no more than:

$$\sum_{j=0}^{d} 2(d-j)2^j$$

Proof that Heapify runs in $O(n)$ time

- Suppose the tree has n nodes and d levels (so $2^d \leq n < 2^{d+1}$).
- If node i is at level j, `SiftDown(H,i)` needs $\leq 2(d-j)$ comparisons.
- There are at most 2^j nodes at level j.
- So total number of comparisons is no more than:

$$\sum_{j=0}^{d} 2(d-j)2^j \quad = \quad 2d\sum_{j=0}^{d} 2^j - 2\sum_{j=0}^{d} j2^j$$

Proof that Heapify runs in $O(n)$ time

- Suppose the tree has n nodes and d levels (so $2^d \leq n < 2^{d+1}$).
- If node i is at level j, `SiftDown(H,i)` needs $\leq 2(d-j)$ comparisons.
- There are at most 2^j nodes at level j.
- So total number of comparisons is no more than:

$$\begin{aligned}\sum_{j=0}^{d} 2(d-j)2^j &= 2d\sum_{j=0}^{d} 2^j - 2\sum_{j=0}^{d} j2^j \\ &= 2d(2^{d+1}-1) - 2\left[(d-1)2^{d+1}+2\right]\end{aligned}$$

Proof that Heapify runs in $O(n)$ time

- Suppose the tree has n nodes and d levels (so $2^d \le n < 2^{d+1}$).
- If node i is at level j, `SiftDown(H,i)` needs $\le 2(d-j)$ comparisons.
- There are at most 2^j nodes at level j.
- So total number of comparisons is no more than:

$$
\begin{aligned}
\sum_{j=0}^{d} 2(d-j)2^j &= 2d\sum_{j=0}^{d} 2^j - 2\sum_{j=0}^{d} j2^j \\
&= 2d(2^{d+1}-1) - 2\left[(d-1)2^{d+1}+2\right] \\
&= 2d2^{d+1} - 2d - 2d2^{d+1} + 2\cdot 2^{d+1} - 4
\end{aligned}
$$

Proof that Heapify runs in $O(n)$ time

- Suppose the tree has n nodes and d levels (so $2^d \leq n < 2^{d+1}$).
- If node i is at level j, `SiftDown(H,i)` needs $\leq 2(d-j)$ comparisons.
- There are at most 2^j nodes at level j.
- So total number of comparisons is no more than:

$$
\begin{aligned}
\sum_{j=0}^{d} 2(d-j)2^j &= 2d\sum_{j=0}^{d} 2^j - 2\sum_{j=0}^{d} j2^j \\
&= 2d(2^{d+1}-1) - 2\left[(d-1)2^{d+1}+2\right] \\
&= 2d2^{d+1} - 2d - 2d2^{d+1} + 2\cdot 2^{d+1} - 4 \\
&= 4\cdot 2^d - 2d - 4
\end{aligned}
$$

Proof that Heapify runs in $O(n)$ time

- Suppose the tree has n nodes and d levels (so $2^d \leq n < 2^{d+1}$).
- If node i is at level j, `SiftDown(H,i)` needs $\leq 2(d-j)$ comparisons.
- There are at most 2^j nodes at level j.
- So total number of comparisons is no more than:

$$
\begin{aligned}
\sum_{j=0}^{d} 2(d-j)2^j &= 2d\sum_{j=0}^{d} 2^j - 2\sum_{j=0}^{d} j2^j \\
&= 2d(2^{d+1}-1) - 2\left[(d-1)2^{d+1}+2\right] \\
&= 2d2^{d+1} - 2d - 2d2^{d+1} + 2\cdot 2^{d+1} - 4 \\
&= 4\cdot 2^d - 2d - 4 \\
&< 4\cdot 2^d
\end{aligned}
$$

Proof that Heapify runs in $O(n)$ time

- Suppose the tree has n nodes and d levels (so $2^d \leq n < 2^{d+1}$).
- If node i is at level j, `SiftDown(H,i)` needs $\leq 2(d-j)$ comparisons.
- There are at most 2^j nodes at level j.
- So total number of comparisons is no more than:

$$
\begin{aligned}
\sum_{j=0}^{d} 2(d-j)2^j &= 2d\sum_{j=0}^{d} 2^j - 2\sum_{j=0}^{d} j2^j \\
&= 2d(2^{d+1}-1) - 2\left[(d-1)2^{d+1}+2\right] \\
&= 2d2^{d+1} - 2d - 2d2^{d+1} + 2\cdot 2^{d+1} - 4 \\
&= 4\cdot 2^d - 2d - 4 \\
&< 4\cdot 2^d \leq 4n
\end{aligned}
$$

Proof that Heapify runs in $O(n)$ time

- Suppose the tree has n nodes and d levels (so $2^d \leq n < 2^{d+1}$).
- If node i is at level j, `SiftDown(H,i)` needs $\leq 2(d-j)$ comparisons.
- There are at most 2^j nodes at level j.
- So total number of comparisons is no more than:

$$
\begin{aligned}
\sum_{j=0}^{d} 2(d-j)2^j &= 2d\sum_{j=0}^{d} 2^j - 2\sum_{j=0}^{d} j2^j \\
&= 2d(2^{d+1}-1) - 2\left[(d-1)2^{d+1}+2\right] \\
&= 2d2^{d+1} - 2d - 2d2^{d+1} + 2\cdot 2^{d+1} - 4 \\
&= 4\cdot 2^d - 2d - 4 \\
&< 4\cdot 2^d \leq 4n = O(n)
\end{aligned}
$$

Proof that Heapify runs in $O(n)$ time

- Suppose the tree has n nodes and d levels (so $2^d \leq n < 2^{d+1}$).
- If node i is at level j, `SiftDown(H,i)` needs $\leq 2(d-j)$ comparisons.
- There are at most 2^j nodes at level j.
- So total number of comparisons is no more than:

$$
\begin{aligned}
\sum_{j=0}^{d} 2(d-j)2^j &= 2d\sum_{j=0}^{d} 2^j - 2\sum_{j=0}^{d} j2^j \\
&= 2d(2^{d+1}-1) - 2\left[(d-1)2^{d+1}+2\right] \\
&= 2d2^{d+1} - 2d - 2d2^{d+1} + 2\cdot 2^{d+1} - 4 \\
&= 4\cdot 2^d - 2d - 4 \\
&< 4\cdot 2^d \leq 4n = O(n)
\end{aligned}
$$

So heap can be constructed using $O(n)$ comparisons.

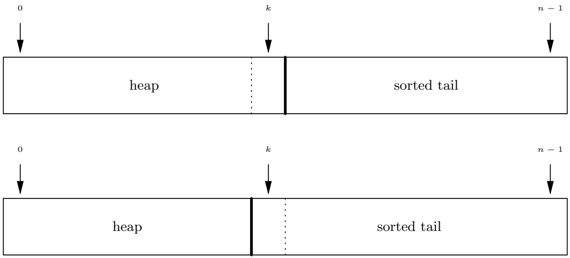
Heapsort: version based on Max Sort

Heapsort: version based on Max Sort

```
def heapsort(A,n):
    heapify(A,n) // form max heap using array A
    for k = n-1 down to 1:
        A[k] = ExtractMax(A)
```

Heapsort: version based on Max Sort

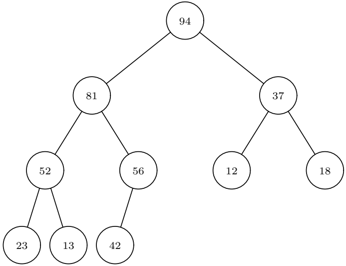
```
def heapsort(A,n):
    heapify(A,n) // form max heap using array A
    for k = n-1 down to 1:
        A[k] = ExtractMax(A)
```



Heapsort example

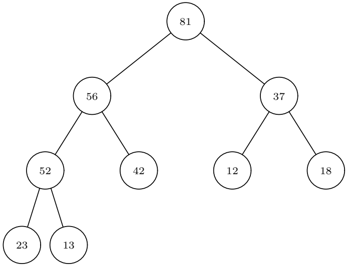
Sort: 13 23 18 94 42 12 37 81 52 56

Heapify:



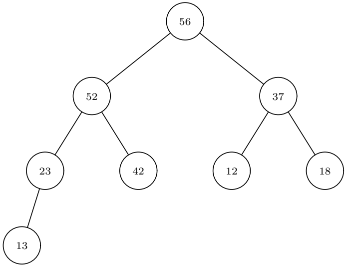
94	81	37	52	56	12	18	23	13	42
0	1	2	3	4	5	6	7	8	9

Heapsort example, continued



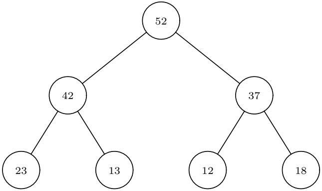
81	56	37	52	42	12	18	23	13	94
0	1	2	3	4	5	6	7	8	9

Heapsort example, continued



56	52	37	23	42	12	18	13	81	94
0	1	2	3	4	5	6	7	8	9

Heapsort example, continued



52	42	37	23	13	12	18	56	81	94
0	1	2	3	4	5	6	7	8	9

Exercise: Finish this example.

Analysis of Heapsort

Analysis of Heapsort

- Storage: $O(1)$ extra space (in place)

Analysis of Heapsort

- Storage: $O(1)$ extra space (in place)
- Time:

Analysis of Heapsort

- Storage: $O(1)$ extra space (in place)
- Time:
  - `Heapify`: $O(n)$

Analysis of Heapsort

- Storage: $O(1)$ extra space (in place)
- Time:
  - `Heapify`: $O(n)$
  - All calls to `ExtractMax`:

$$\sum_{k=1}^{n-1} O\left(\log(k+1)\right) = O(n \log n)$$

Analysis of Heapsort

- Storage: $O(1)$ extra space (in place)
- Time:
  - `Heapify`: $O(n)$
  - All calls to `ExtractMax`:

$$\sum_{k=1}^{n-1} O\left(\log(k+1)\right) = O(n \log n)$$

  - Hence total time is $O(n \log n)$.

Comparison-based sorts: Summary/Comparison

Comparison-based sorts: Summary/Comparison

Sort	Worst-case Time	Storage Requirement	Remarks
Insertion Sort	$O(n^2)$	In-place	Good if input is almost sorted.
QuickSort	$O(n^2)$	$O(\log n)$ extra for stack	$O(n \log n)$ expected time.
Mergesort	$O(n \log n)$	$O(n)$ extra for merge	
Heapsort	$O(n \log n)$	In-place	Can output k smallest in sorted order in $O(n + k \log n)$ time.

Lower bound on comparison-based sorting

Lower bound on comparison-based sorting

- Based on Decision Tree model.

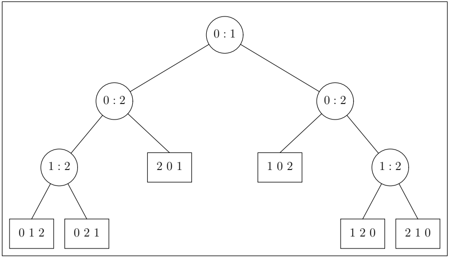
Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size n using comparisons can be modeled as a decision tree:

Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size n using comparisons can be modeled as a decision tree:

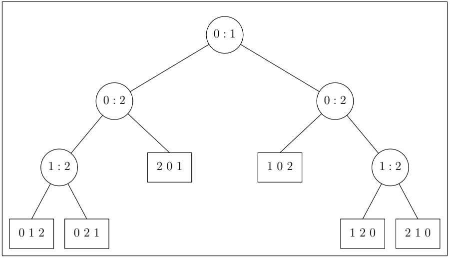
Example: Decision tree for sorting 3 items



Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size n using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.

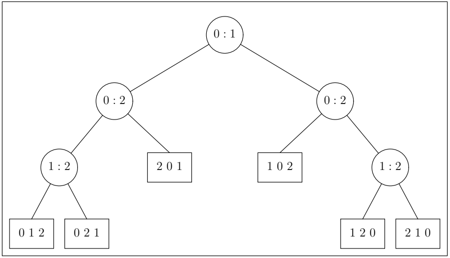
Example: Decision tree for sorting 3 items



Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size n using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i : j$ describes for what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).

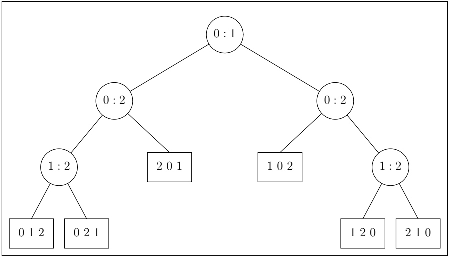
Example: Decision tree for sorting 3 items



Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size n using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i : j$ describes for what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).
  - Each leaf node is a permutation of $0, \ldots n-1$.

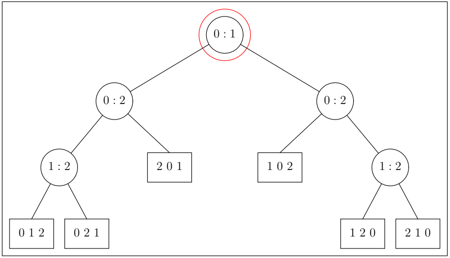
Example: Decision tree for sorting 3 items



Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size n using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i : j$ describes for what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).
  - Each leaf node is a permutation of $0, \ldots n-1$.

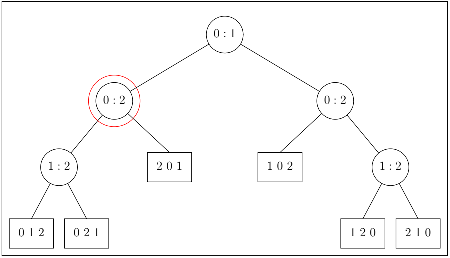
Example: Decision tree for sorting 3 items



Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size n using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i : j$ describes for what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).
  - Each leaf node is a permutation of $0, \ldots n-1$.

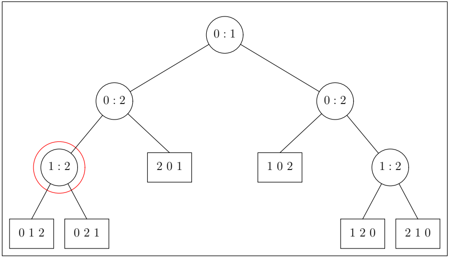
Example: Decision tree for sorting 3 items



Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size n using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i : j$ describes for what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).
  - Each leaf node is a permutation of $0, \ldots n-1$.

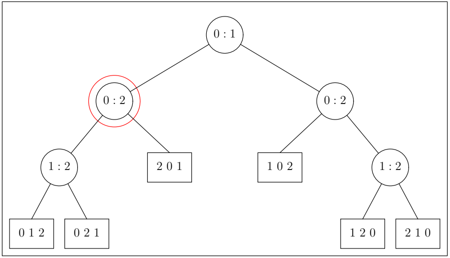
Example: Decision tree for sorting 3 items



Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size n using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i : j$ describes for what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).
  - Each leaf node is a permutation of $0, \ldots n-1$.

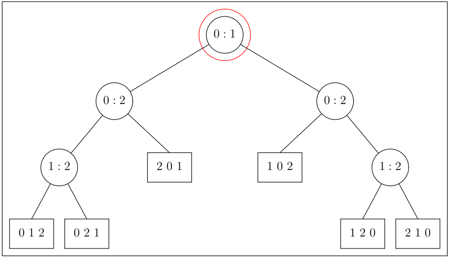
Example: Decision tree for sorting 3 items



Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size n using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i : j$ describes for what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).
  - Each leaf node is a permutation of $0, \ldots n-1$.

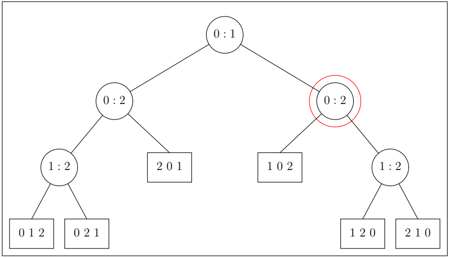
Example: Decision tree for sorting 3 items



Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size n using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i : j$ describes for what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).
  - Each leaf node is a permutation of $0, \ldots n-1$.

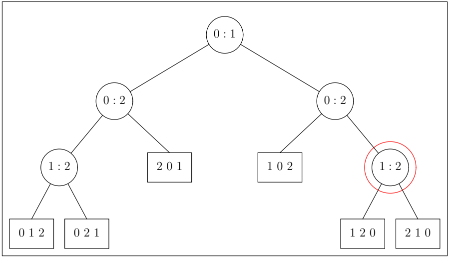
Example: Decision tree for sorting 3 items



Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size n using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i : j$ describes for what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).
  - Each leaf node is a permutation of $0, \ldots n-1$.

Example: Decision tree for sorting 3 items



Lower bound on comparison-based sorting (continued)

Lower bound on comparison-based sorting (continued)

1. Any comparison-based algorithm for sorting a list of size n can be modeled by a decision tree with at least $n!$ leaf nodes.

Lower bound on comparison-based sorting (continued)

1. Any comparison-based algorithm for sorting a list of size n can be modeled by a decision tree with at least $n!$ leaf nodes.
2. Since the decision tree is a binary tree with $n!$ leaves, the depth is at least $\lceil \lg n! \rceil$.

Lower bound on comparison-based sorting (continued)

1. Any comparison-based algorithm for sorting a list of size n can be modeled by a decision tree with at least $n!$ leaf nodes.
2. Since the decision tree is a binary tree with $n!$ leaves, the depth is at least $\lceil \lg n! \rceil$.
3. The worst-case number of comparisons for the algorithm is the depth of the decision tree.

Lower bound on comparison-based sorting (continued)

1. Any comparison-based algorithm for sorting a list of size n can be modeled by a decision tree with at least $n!$ leaf nodes.
2. Since the decision tree is a binary tree with $n!$ leaves, the depth is at least $\lceil \lg n! \rceil$.
3. The worst-case number of comparisons for the algorithm is the depth of the decision tree.
4. $\lg n! = \Omega(n \log n)$ (proof on next slide)

Lower bound on comparison-based sorting (continued)

1. Any comparison-based algorithm for sorting a list of size n can be modeled by a decision tree with at least $n!$ leaf nodes.
2. Since the decision tree is a binary tree with $n!$ leaves, the depth is at least $\lceil \lg n! \rceil$.
3. The worst-case number of comparisons for the algorithm is the depth of the decision tree.
4. $\lg n! = \Omega(n \log n)$ (proof on next slide)

Fact #2 and Fact #3 imply an exact bound:

Lower bound on comparison-based sorting (continued)

1. Any comparison-based algorithm for sorting a list of size n can be modeled by a decision tree with at least $n!$ leaf nodes.
2. Since the decision tree is a binary tree with $n!$ leaves, the depth is at least $\lceil \lg n! \rceil$.
3. The worst-case number of comparisons for the algorithm is the depth of the decision tree.
4. $\lg n! = \Omega(n \log n)$ (proof on next slide)

Fact #2 and Fact #3 imply an exact bound:

Any comparison-based algorithm for sorting a list of size n must perform at least $\lceil \lg n! \rceil$ comparisons in the worst case.

Lower bound on comparison-based sorting (continued)

1. Any comparison-based algorithm for sorting a list of size n can be modeled by a decision tree with at least $n!$ leaf nodes.
2. Since the decision tree is a binary tree with $n!$ leaves, the depth is at least $\lceil \lg n! \rceil$.
3. The worst-case number of comparisons for the algorithm is the depth of the decision tree.
4. $\lg n! = \Omega(n \log n)$ (proof on next slide)

Fact #2 and Fact #3 imply an exact bound:

Any comparison-based algorithm for sorting a list of size n must perform at least $\lceil \lg n! \rceil$ *comparisons in the worst case.*

The previous statement and Fact #4 imply an asymptotic bound:

Lower bound on comparison-based sorting (continued)

1. Any comparison-based algorithm for sorting a list of size n can be modeled by a decision tree with at least $n!$ leaf nodes.
2. Since the decision tree is a binary tree with $n!$ leaves, the depth is at least $\lceil \lg n! \rceil$.
3. The worst-case number of comparisons for the algorithm is the depth of the decision tree.
4. $\lg n! = \Omega(n \log n)$ (proof on next slide)

Fact #2 and Fact #3 imply an exact bound:

Any comparison-based algorithm for sorting a list of size n must perform at least $\lceil \lg n! \rceil$ comparisons in the worst case.

The previous statement and Fact #4 imply an asymptotic bound:

Any comparison-based algorithm for sorting a list of size n must perform at least $\Omega(n \log n)$ comparisons in the worst case.

Lower bound on comparison-based sorting (continued)

Lower bound on comparison-based sorting (continued)

Proof that $\lg n! = \Omega(n \log n)$:

Lower bound on comparison-based sorting (continued)

Proof that $\lg n! = \Omega(n \log n)$:

$$n! = n \cdot (n-1) \cdot (n-3) \cdots 2 \cdot 1$$

Lower bound on comparison-based sorting (continued)

Proof that $\lg n! = \Omega(n \log n)$:

$$n! = n \cdot (n-1) \cdot (n-3) \cdots 2 \cdot 1$$

The first $\lceil n/2 \rceil$ terms in the product are all $\geq \left\lceil \frac{n}{2} \right\rceil$.

Lower bound on comparison-based sorting (continued)

Proof that $\lg n! = \Omega(n \log n)$:

$$n! = n \cdot (n-1) \cdot (n-3) \cdots 2 \cdot 1$$

The first $\lceil n/2 \rceil$ terms in the product are all $\geq \left\lceil \frac{n}{2} \right\rceil$.

This implies:

Lower bound on comparison-based sorting (continued)

Proof that $\lg n! = \Omega(n \log n)$:

$$n! = n \cdot (n-1) \cdot (n-3) \cdots 2 \cdot 1$$

The first $\lceil n/2 \rceil$ terms in the product are all $\geq \left\lceil \frac{n}{2} \right\rceil$.

This implies:

$$n! \geq \left\lceil \frac{n}{2} \right\rceil^{\left\lceil \frac{n}{2} \right\rceil}$$

Lower bound on comparison-based sorting (continued)

Proof that $\lg n! = \Omega(n \log n)$:

$$n! = n \cdot (n-1) \cdot (n-3) \cdots 2 \cdot 1$$

The first $\lceil n/2 \rceil$ terms in the product are all $\geq \left\lceil \frac{n}{2} \right\rceil$.

This implies:

$$n! \geq \left\lceil \frac{n}{2} \right\rceil^{\left\lceil \frac{n}{2} \right\rceil} \geq \left(\frac{n}{2} \right)^{\frac{n}{2}}$$

Lower bound on comparison-based sorting (continued)

Proof that $\lg n! = \Omega(n \log n)$:

$$n! = n \cdot (n-1) \cdot (n-3) \cdots 2 \cdot 1$$

The first $\lceil n/2 \rceil$ terms in the product are all $\geq \left\lceil \frac{n}{2} \right\rceil$.

This implies:

$$n! \geq \left\lceil \frac{n}{2} \right\rceil^{\left\lceil \frac{n}{2} \right\rceil} \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

Take $\log_2$ of both sides:

Lower bound on comparison-based sorting (continued)

Proof that $\lg n! = \Omega(n \log n)$:

$$n! = n \cdot (n-1) \cdot (n-3) \cdots 2 \cdot 1$$

The first $\lceil n/2 \rceil$ terms in the product are all $\geq \left\lceil \frac{n}{2} \right\rceil$.

This implies:

$$n! \geq \left\lceil \frac{n}{2} \right\rceil^{\left\lceil \frac{n}{2} \right\rceil} \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

Take $\log_2$ of both sides:

$$\lg n! \geq \left(\frac{n}{2}\right) \lg \left(\frac{n}{2}\right)$$

Lower bound on comparison-based sorting (continued)

Proof that $\lg n! = \Omega(n \log n)$:

$$n! = n \cdot (n-1) \cdot (n-3) \cdots 2 \cdot 1$$

The first $\lceil n/2 \rceil$ terms in the product are all $\geq \lceil \frac{n}{2} \rceil$.

This implies:

$$n! \geq \left\lceil \frac{n}{2} \right\rceil^{\left\lceil \frac{n}{2} \right\rceil} \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

Take $\log_2$ of both sides:

$$\lg n! \geq \left(\frac{n}{2}\right) \lg \left(\frac{n}{2}\right) = \left(\frac{n}{2}\right) (\lg n - 1)$$

Lower bound on comparison-based sorting (continued)

Proof that $\lg n! = \Omega(n \log n)$:

$$n! = n \cdot (n-1) \cdot (n-3) \cdots 2 \cdot 1$$

The first $\lceil n/2 \rceil$ terms in the product are all $\geq \left\lceil \frac{n}{2} \right\rceil$.

This implies:

$$n! \geq \left\lceil \frac{n}{2} \right\rceil^{\left\lceil \frac{n}{2} \right\rceil} \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

Take $\log_2$ of both sides:

$$\lg n! \geq \left(\frac{n}{2}\right) \lg \left(\frac{n}{2}\right) = \left(\frac{n}{2}\right)(\lg n - 1) = \Omega(n \lg n)$$

Asymptotic optimality of MergeSort and HeapSort

We have just shown:

Asymptotic optimality of MergeSort and HeapSort

We have just shown:

> *Any comparison-based algorithm for sorting a list of size n must perform at least $\Omega(n \log n)$ comparisons in the worst case.*

Asymptotic optimality of MergeSort and HeapSort

We have just shown:

Any comparison-based algorithm for sorting a list of size n must perform at least $\Omega(n \log n)$ comparisons in the worst case.

Earlier we showed:

Asymptotic optimality of MergeSort and HeapSort

We have just shown:

Any comparison-based algorithm for sorting a list of size n must perform at least $\Omega(n \log n)$ comparisons in the worst case.

Earlier we showed:

The worst-case running time of MergeSort and HeapSort on an input of size n is $O(n \log n)$.

Asymptotic optimality of MergeSort and HeapSort

We have just shown:

Any comparison-based algorithm for sorting a list of size n must perform at least $\Omega(n \log n)$ comparisons in the worst case.

Earlier we showed:

The worst-case running time of MergeSort and HeapSort on an input of size n is $O(n \log n)$.

Conclusions:

Asymptotic optimality of MergeSort and HeapSort

We have just shown:

Any comparison-based algorithm for sorting a list of size n must perform at least $\Omega(n \log n)$ comparisons in the worst case.

Earlier we showed:

The worst-case running time of MergeSort and HeapSort on an input of size n is $O(n \log n)$.

Conclusions:

1. MergeSort and HeapSort are asymptotically optimal.

Asymptotic optimality of MergeSort and HeapSort

We have just shown:

Any comparison-based algorithm for sorting a list of size n must perform at least $\Omega(n \log n)$ comparisons in the worst case.

Earlier we showed:

The worst-case running time of MergeSort and HeapSort on an input of size n is $O(n \log n)$.

Conclusions:

1. MergeSort and HeapSort are asymptotically optimal.
2. The lower bound is asymptotically tight (i.e., cannot be improved asymptotically)