Heap Sort
Heapsort

Heap Sort Algorithm (build + sort)

6 10 1 4 7 9 3 2 8 11

Build Heap (Max)

11 10 9 8 7 1 3 2 4 6

Sort Max Heap

1 2 3 4 6 7 8 9 10 11
Heapsort Algorithm

Function Heapsort(A)

1  #Create max heap
    Build_Max_Heap from unordered array A

2  # Finish sorting
    for i = n downto 2 do
        discard node i from heap (decrement heap size)
        sift(A[1:i-1], 1) because new root may violate max heap property
Build Max Heap

Function Build_Max_Heap(A)
    set heap size to the length of the array
    for j = n/2 down to 1 do
        sift(A, j)

20, 8, 7, 9, 6, 54

20, 8, 54, 9, 6, 7
The root of the tree is A[1], and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

```
function parent(i)
    return i/2

function left(i)
    return 2*i

function right(i)
    return 2*i + 1
```
Max-Heapify (sift)

```python
function sift(arr, i)
  n ← len(arr)
  l ← left(i)
  r ← right(i)
  if l <= n and arr[l] > arr[i] then
    largest ← l
  else
    largest ← i
  if r <= n and arr[r] > arr[largest] then
    largest ← r
  if largest != i then
    arr[i] ← arr[largest]
    sift(arr, largest)
  return arr
```
Start with an array (it is not a max heap)
Function Build_Max_Heap(A)
    set heap size to the length of the array
    for j= n/2 down to 1 do
        sift(A, j)

function sift(arr,i)
    n ← len(arr)
    l ← left(i)
    r ← right(i)

    if l <= n and arr[l] > arr[i] then
        largest ← l
    else
        largest ← i

    if r <= n and arr[r] > arr[largest] then
        largest = r

    if largest != i then
        arr[i] ↔ arr[largest]
        sift(arr,largest)
    return arr

function parent(i)
    return i/2

function left(i)
    return 2*i

function right(i)
    return 2*i + 1
Function Build_Max_Heap(A)
   set heap size to the length of the array
   for j = n/2 down to 1 do
       sift(A, j)
Function Build_Max_Heap(A)
    set heap size to the length of the array
    for j= n/2 down to 1 do
        sift(A, j)
Function Build_Max_Heap(A)
   set heap size to the length of the array
   for j = n/2 down to 1 do
      sift(A, j)
Exchange 6 and 11

Function Build_Max_Heap(A)
   set heap size to the length of the array
   for j = n/2 down to 1 do
      sift(A, j)

function sift(arr, i)
   n ← len(arr)
   l ← left(i)
   r ← right(i)
   if l ≤ n and arr[l] > arr[i] then
      largest ← l
   else
      largest ← i
   if r ≤ n and arr[r] > arr[largest] then
      largest ← r
   if largest ≠ i then
      arr[i] ← arr[largest]
      sift(arr, largest)
   return arr
Exchange 6 and 10

Function Build_Max_Heap(A)
  set heap size to the length of the array
  for j= n/2 down to 1 do
    sift(A, j)

  function sift(arr,i)
    n ← len(arr)
    l ← left(i)
    r ← right(i)
    if l <= n and arr[l] > arr[i] then
      largest ← l
    else
      largest ← i
    if r <= n and arr[r] > arr[largest] then
      largest ← r
    if largest != i then
      arr[i] ← arr[largest]
      sift(arr,largest)
  return arr
Function Build_Max_Heap(A)
   set heap size to the length of the array
   for j = n/2 down to 1 do
      sift(A, j)

function sift(arr,i)
   n ← len(arr)
l ← left(i)
r ← right(i)
if l ≤ n and arr[l] > arr[i] then
   largest ← l
else
   largest ← i
if r ≤ n and arr[r] > arr[largest] then
   largest ← r
if largest != i then
   arr[i] ↔ arr[largest]
sift(arr,largest)
return arr

Exchange 6 and 7

<table>
<thead>
<tr>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>6</th>
<th>1</th>
<th>3</th>
<th>2</th>
<th>4</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>i</td>
<td>largest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
max_heapify

Function Build_Max_Heap(A)
    set heap size to the length of the array
    for j = n/2 down to 1 do
        sift(A, j)

function sift(arr, i)
    n ← len(arr)
    l ← left(i)
    r ← right(i)
    if l ≤ n and arr[l] > arr[i] then
        largest ← l
    else
        largest ← i
    if r ≤ n and arr[r] > arr[largest] then
        largest ← r
    if largest ≠ i then
        arr[i] ↔ arr[largest]
        sift(arr, largest)
    return arr

11 10 9 8 7 1 3 2 4 6

j i largest
Function Heapsort(A)

# Create max heap
Build_Max_Heap from unordered array A

# Finish sorting
for i = n downto 2 do
    discard node i from heap (decrement heap size)
    sift(A[1:i-1], 1) because new root may violate max heap property
Function Heapsort(A)
# Create max heap
Build_Max_Heap from unordered array A
# Finish sorting
for i = n downto 2 do
    discard node i from heap (decrement heap size)
    sift(A[1:i-1], 1) because new root may violate max heap property
Remove 11 from the heap
Swap 6 and 10

function sift(arr, i)
  n ← len(arr)
  l ← left(i)
  r ← right(i)
  if l ≤ n and arr[l] > arr[i] then
    largest ← l
  else
    largest ← i
  if r ≤ n and arr[r] > arr[largest] then
    largest ← r
  if largest ≠ i then
    arr[i] ← arr[largest]
    sift(arr, largest)
  return arr
Exchange 6 and 8
Function Heapsort(A)

#Create max heap
Build_Max_Heap from unordered array A

# Finish sorting
for i = n downto 2 do
discard node i from heap (decrement heap size)
sift(A[1:i-1],1) because new root may violate max heap property
Remove 10 from the heap
Exchange 4 and 9
Function Heapsort(A)

# Create max heap
Build_Max_Heap from unordered array A

# Finish sorting
for i = n downto 2 do
    discard node i from heap (decrement heap size)
    sift(A[1:i-1],1) because new root may violate max heap property
Remove 9 from the heap
Exchange 2 and 8
Exchange 2 and 7
Function Heapsort(A)

# Create max heap
Build_Max_Heap from unordered array A

# Finish sorting
for i = n downto 2 do
  discard node i from heap (decrement heap size)
  sift(A[1:i-1],1) because new root may violate max heap property
Remove 8 from the heap
Exchange 3 and 7
Exchange 3 and 6

[Diagram of a tree with nodes labeled 1 to 11, indicating a specific exchange of nodes 3 and 6.]
Function Heapsort(A)
#Create max heap
Build_Max_Heap from unordered array A
# Finish sorting
for i = n downto 2 do
discard node i from heap (decrement heap size)
sift(A[1:i-1],1) because new root may violate max heap property
Remove 7 from the heap
Exchange 1 and 6
Exchange 1 and 3
Exchange 6 and 2 and remove from the heap

Function Heapsort(A)

# Create max heap
Build_Max_Heap from unordered array A
# Finish sorting
for i = n downto 2 do
    discard node i from heap (decrement heap size)
    sift(A[1:i-1], 1) because new root may violate max heap property
Exchange 4 and 2
Function Heapsort(A)

# Create max heap
Build_Max_Heap from unordered array A

# Finish sorting
for i = n downto 2 do
    discard node i from heap (decrement heap size)
    sift(A[1:i-1], 1) because new root may violate max heap property
Remove 4, exchange 1 and 3
Function Heapsort(A)

# Create max heap
Build_Max_Heap from unordered array A

# Finish sorting
for i = n downto 2 do


discard node i from heap (decrement heap size)

sift(A[1:i-1],1) because new root may violate max heap property
Function Heapsort(A)

# Create max heap
Build_Max_Heap from unordered array A

# Finish sorting
for i = n downto 2 do
discard node i from heap (decrement heap size)
sift(A[1:i-1], 1) because new root may violate max heap property
The array is sorted

1
2
3
4
6
7
8
9
10
11

1 2 3 4 6 7 8 9 10 11
Heap Sort Algorithm (build + sort)

Build Heap (Max)

Sort Max Heap

Sorted Output

6 10 1 4 7 9 3 2 8 11

1 2 3 4 6 7 8 9 10 11
Heapsort Algorithm

Function Heapsort(A)

# Create max heap
Build_Max_Heap from unordered array A

# Finish sorting
for i = n downto 2 do
    discard node i from heap (decrement heap size)
    sift(A[1:i-1], 1) because new root may violate max heap property
Function Build_Max_Heap(A)
    set heap size to the length of the array
    for j = n/2 down to 1 do
        sift(A, j)
Heap

- The root of the tree is A[1], and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

  ```python
  function parent(i)
      return i/2
  
  function left(i)
      return 2*i
  
  function right(i)
      return 2 *i + 1
  ```
Max-Heapify (sift)

```python
function sift(arr, i)
    n ← len(arr)
    l ← left(i)
    r ← right(i)

    if l <= n and arr[l] > arr[i] then
        largest ← l
    else:
        largest ← i

    if r <= n and arr[r] > arr[largest] then
        largest ← r

    if largest != i then
        arr[i] ← arr[largest]
        sift(arr, largest)

    return arr
```