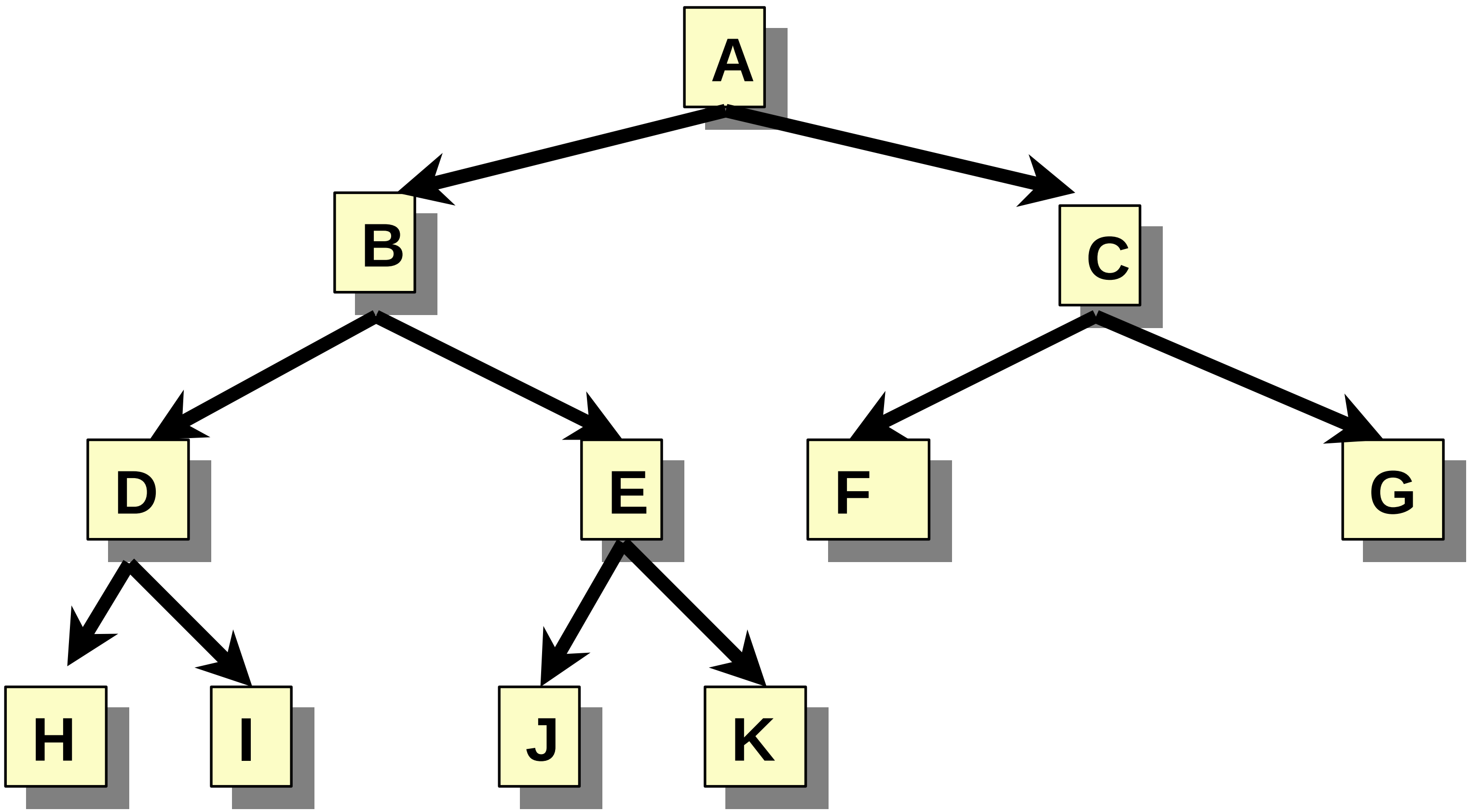
Heap

Complete Binary Tree

- A **complete binary tree** is a binary tree in which every level, *except possibly the last*, is completely filled, and all nodes are as far left as possible.

Complete Binary Trees - Example

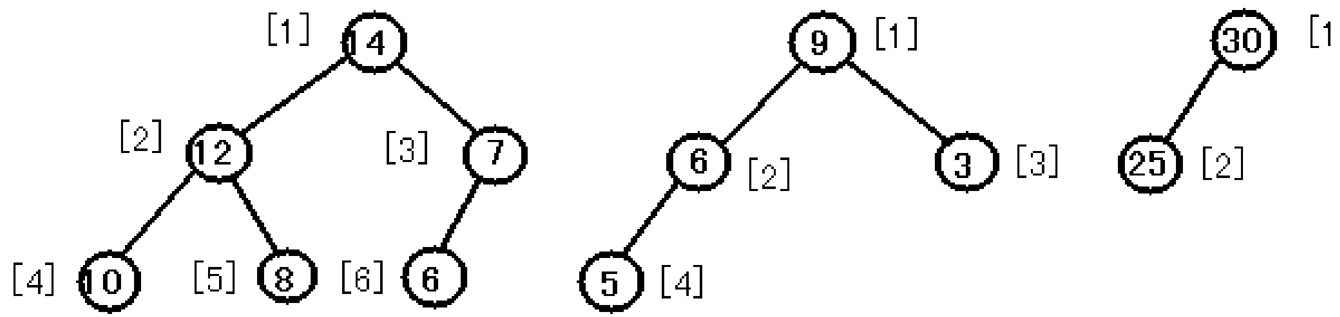


Heap

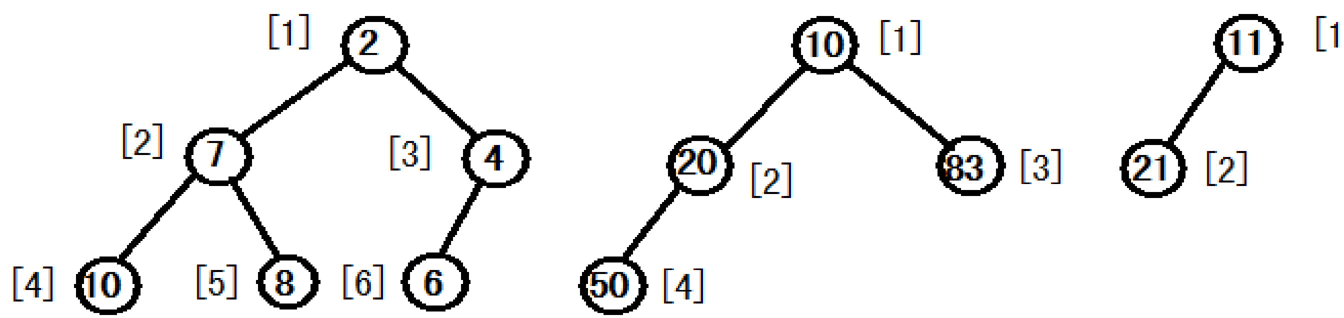
- A special form of complete binary tree
- Max-Heap: root node has the largest key
  - It is a tree in which the key value in each node is no smaller than the key values in its children
- Min-Heap: root node has the smallest key
  - It is a tree in which the key value in each node is no larger than the key values in its children

Heap

- Example:
  - Max-Heap

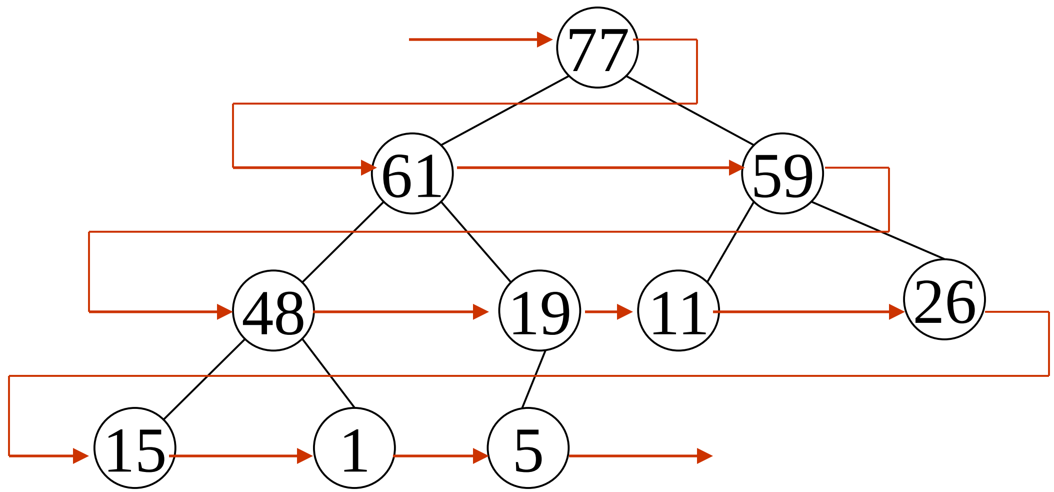


  - Min-Heap



Heap implementation

- Notice:
  - Heap data structure is a complete binary tree! (**Nice representation in Array**)
  - Heap using an array of memory.



  - Stored using array

index	1	2	3	4	5	6	7	8	9	10
value	77	61	59	48	19	11	26	15	1	5

Heap operations

- Operations
  - Creation of an empty heap
  - Insertion of a new element into the heap
  - Deletion of the largest(smallest) element from the heap

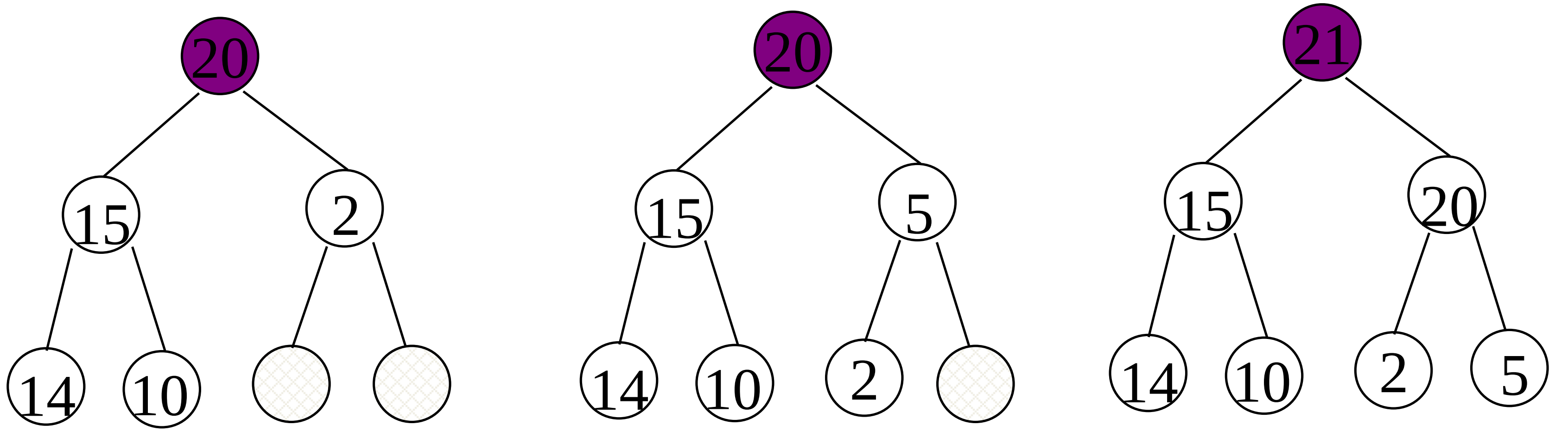
- Heap is complete binary tree, can be represented by array. So the complexity of inserting any node or deleting the root node from Heap is O(height)= $O(\log_2 n)$

Heap

Given the index i of a node

- Parent(i)
  - return i/2
- LeftChild(i)
  - return 2i
- RightChild(i)
  - Return 2i+1

Example of Insertion to Max Heap



initial location of new node

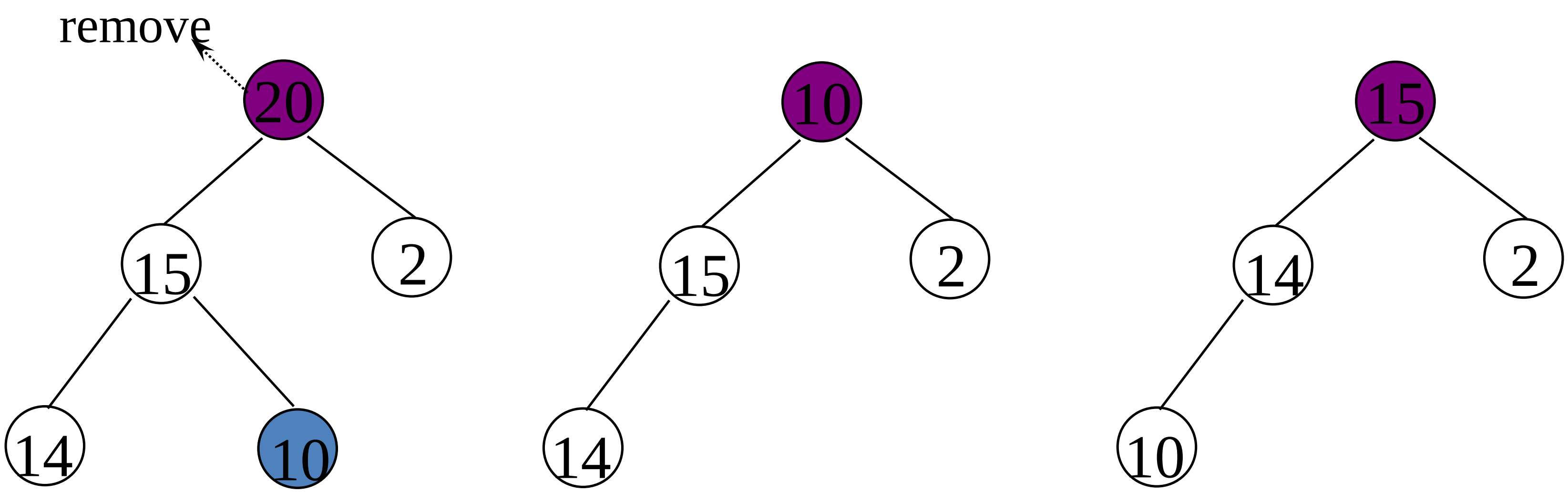
insert 5 into heap

insert 21 into heap

Insertion into a Max Heap

```
void insert_max_heap(element item, int size, int maxsize)
{
  int i;
  if (size+1 >= maxsize)
  { printf("the heap is full.\n");
    exit(1);
  }
  ++size;
  i=size;
  while ((i!=1)&&(item.key>heap[i/2].key))
  { heap[i] = heap[i/2];
    i = i / 2;
  }
  heap[i]= item;
}
```

Example of Deletion from Max Heap



Deletion from a Max Heap

```
element delete_max_heap(int size, int maxsize)
{
  int parent, child;
  element item, temp;
  if (size==0) {
    printf("The heap is empty");
    exit(1);
  }
  /* save value of the element with the highest key */
  item = heap[1];
  /* use last element in heap to adjust heap */
  heap[1] = heap[size];
  --size;
```

Deletion from a Max Heap

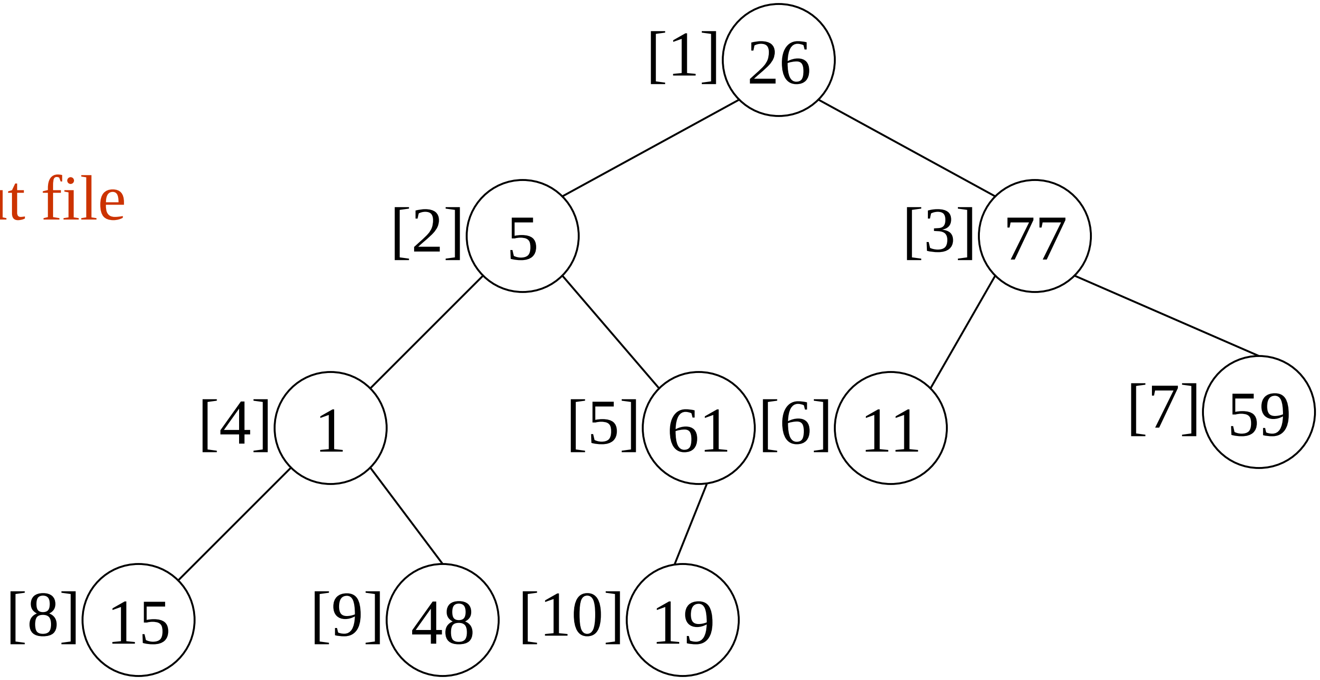
```
temp=heap[1]; parent=1; child=2;
while (child <= size)
  {
    /* find the larger child of the current parent */
    if ((child+1 <= size)&&
       (heap[child].key < heap[child+1].key))
       child++;
    if (heap[parent].key >= heap[child].key) break;
    /* move to the next lower level */
    heap[parent] = heap[child];
    parent = child
    child *= 2;
  }
  heap[parent] = temp;
  return item;
}
```

Application of MaxHeap: Heap Sort

- See an illustration first
  - Array interpreted as a binary tree

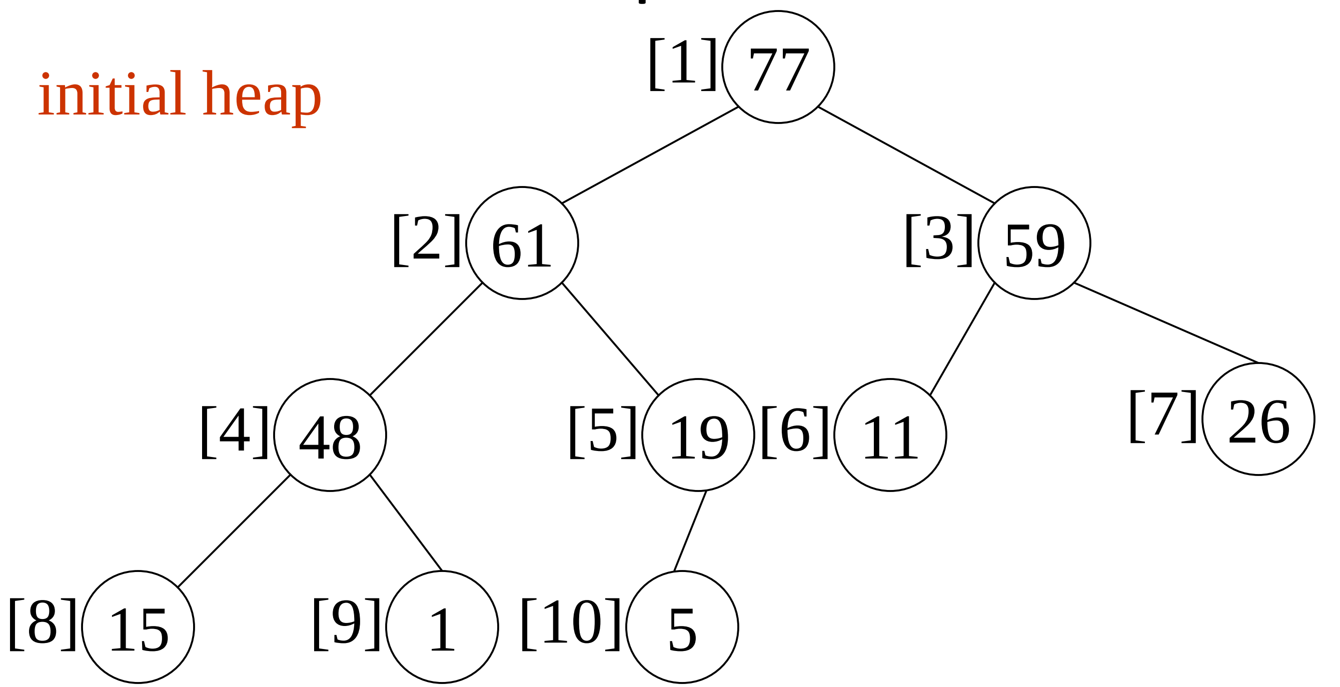
1	2	3	4	5	6	7	8	9	10
26	5	77	1	61	11	59	15	48	19

input file



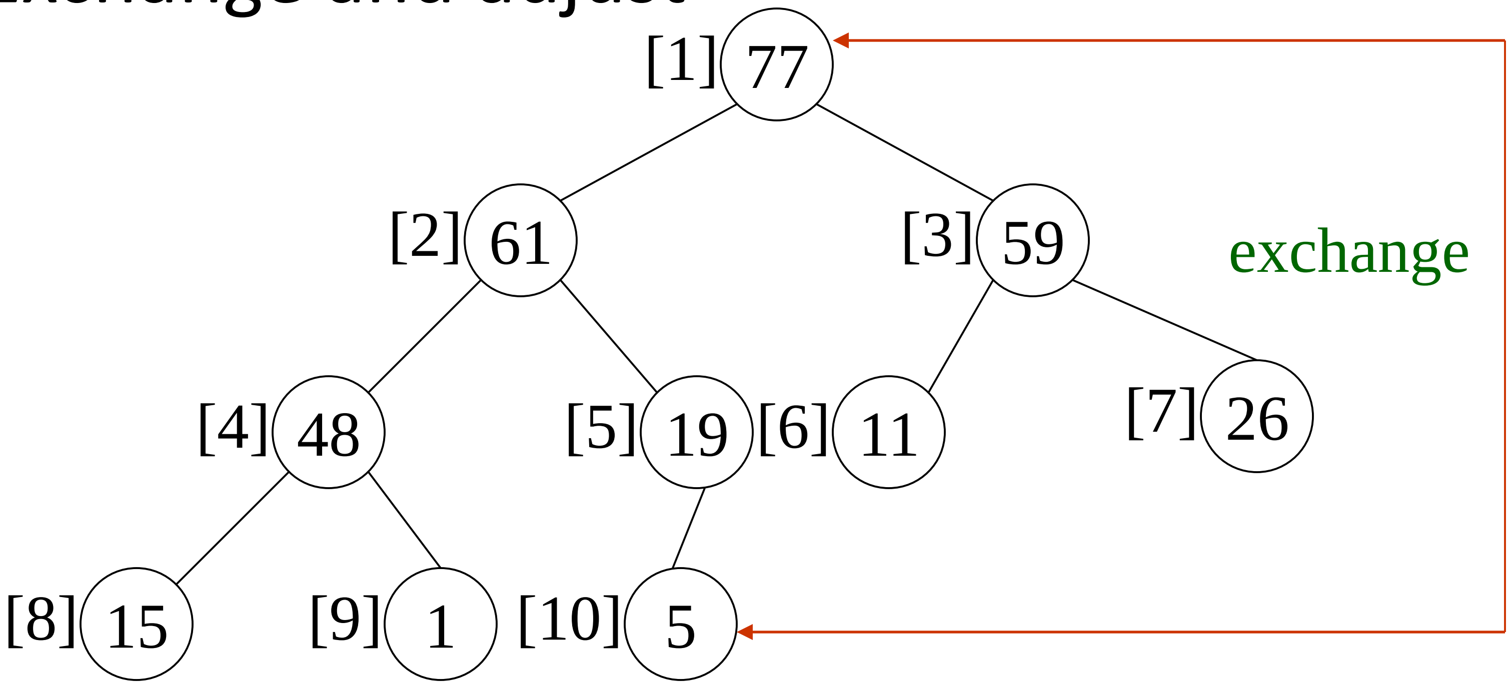
Heap Sort

- Adjust it to a MaxHeap

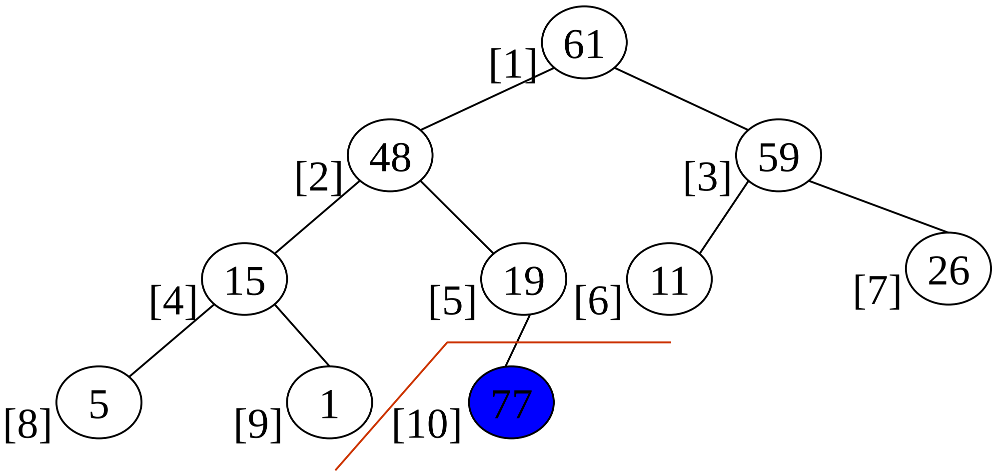


Heap Sort

- Exchange and adjust

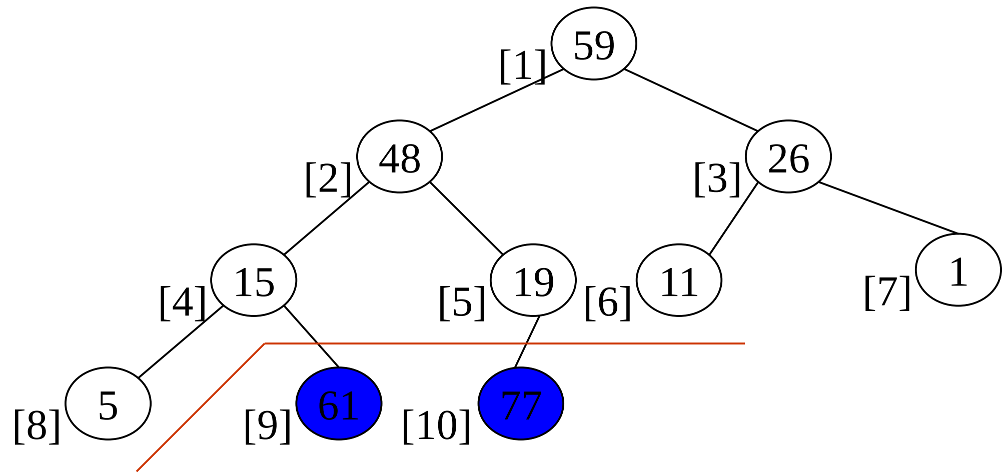


Heap Sort



[1]
61
[2]
48
[3]
59
[4]
15
[5]
19
[6]
11
[7]
26
[8]
5
[9]
1
[10]
77

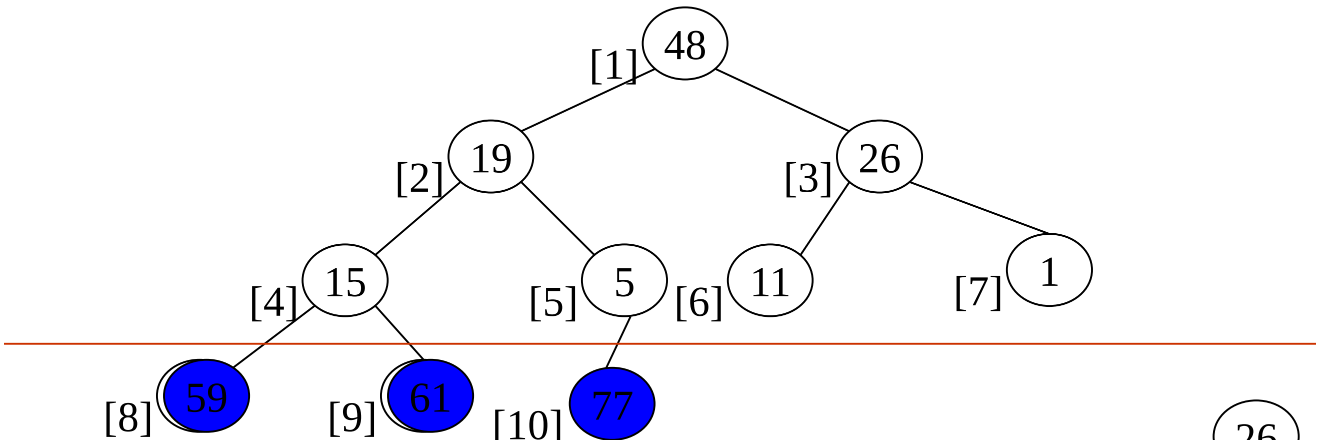
(a)



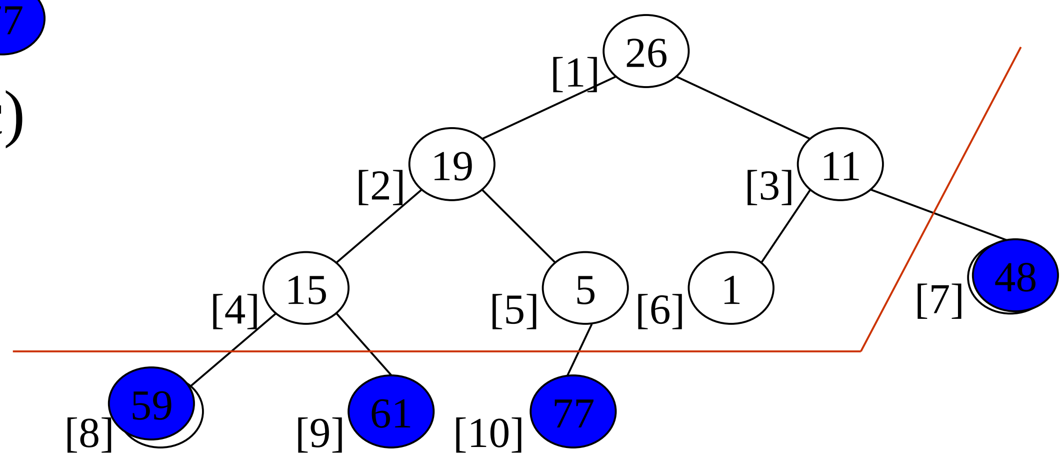
[1]
59
[2]
48
[3]
26
[4]
15
[5]
19
[6]
11
[7]
1
[8]
5
[9]
61
[10]
77

(b)

Heap Sort

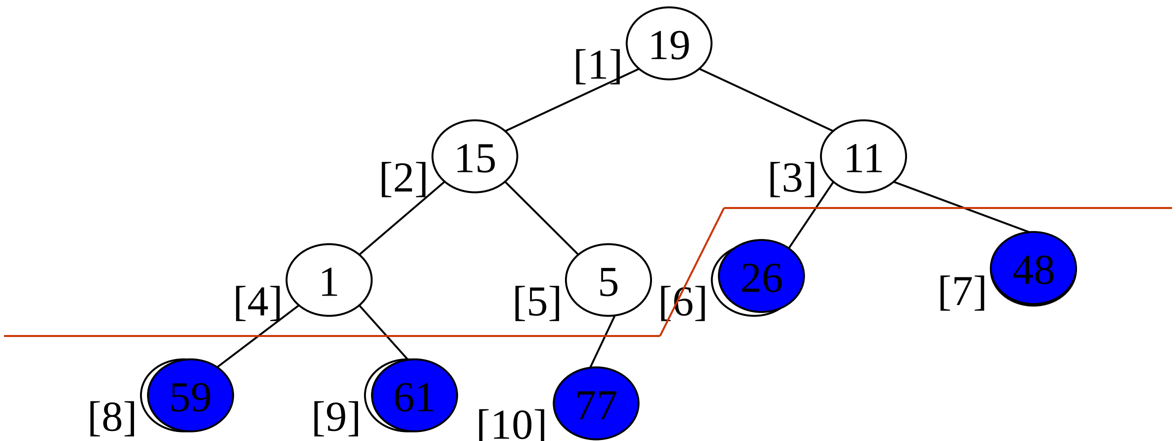

[1] 48
[2] 19
[3] 26
[4] 15
[5] 5
[6] 11
[7] 1
[8] 59
[9] 61
[10] 77

(c)


[1] 26
[2] 19
[3] 11
[4] 15
[5] 5
[6] 1
[7] 48
[8] 59
[9] 61
[10] 77

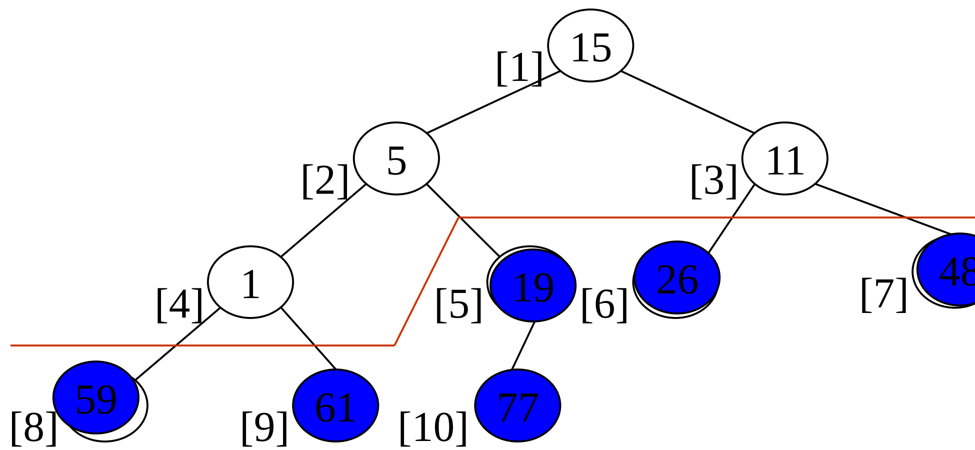
(d)

Heap Sort



[1] 19
[2] 15
[3] 11
[4] 1
[5] 5
[6] 26
[7] 48
[8] 59
[9] 61
[10] 77

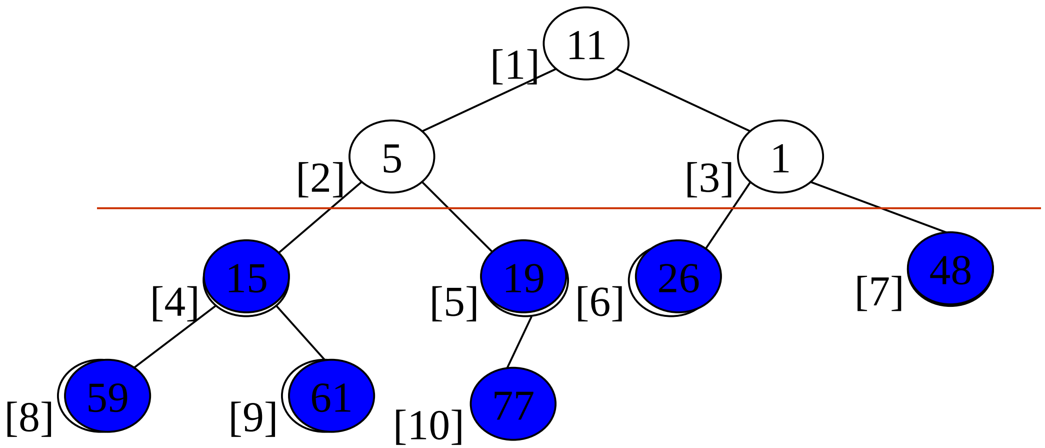
(e)



[1] 15
[2] 5
[3] 11
[4] 1
[5] 19
[6] 26
[7] 48
[8] 59
[9] 61
[10] 77

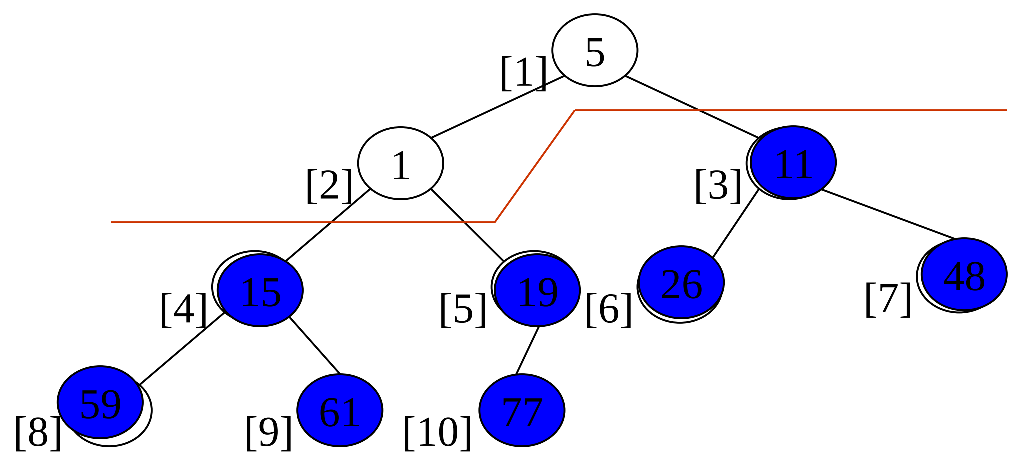
(f)

Heap Sort



11
[1]
5
[2]
1
[3]
15
[4]
19
[5]
26
[6]
48
[7]
59
[8]
61
[9]
77
[10]

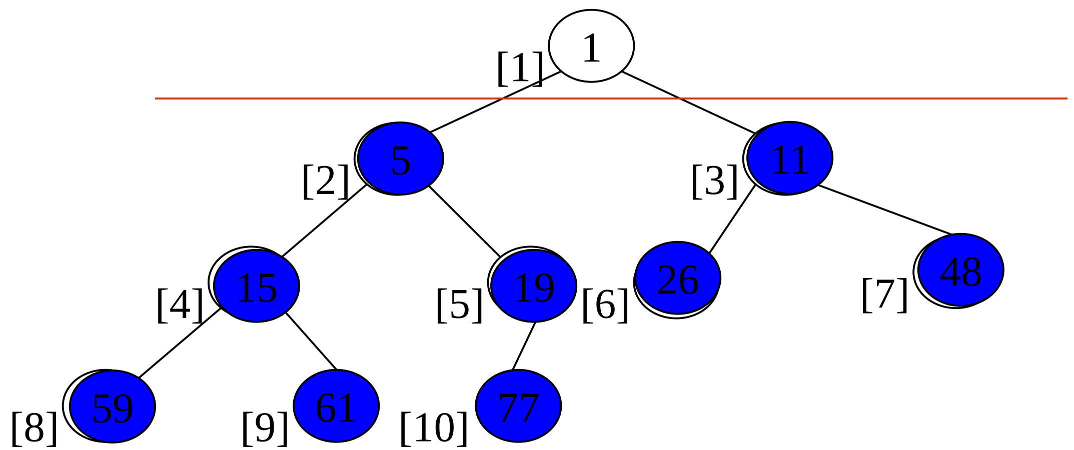
(g)



5
[1]
1
[2]
11
[3]
15
[4]
19
[5]
26
[6]
48
[7]
59
[8]
61
[9]
77
[10]

(h)

Heap Sort



(i)

- So results

77 61 59 48 26 19 15 11 5 1

```
HeapSort(A,size)
{
  BuildMaxHeap(A,size);
  for(i=size downto 2)
  {
     exchange(A[1] <->A[i]);
     size=size-1;
     Heapify(A,1);
   }
}
```

```
BuildMaxHeap(A, size)
{
  for(i=size/2 downto 1)
  {
     Heapify(A,size);
   }
}
```

```
Heapify(A, i)
{
  Lchild=2*i;
  Rchild=Lchild + 1;
if(Lchild≤size and a[Lchild]>A[i])
  then  largest=Lchild;
  else   largest=i;
if(Rchild≤size and a[Rchild]>A[largest])
  then  largest=Rchild;
If(largest≠i)
  then exchange(A[i] <->A[largest]);
         Heapify(A,largest)
}
```