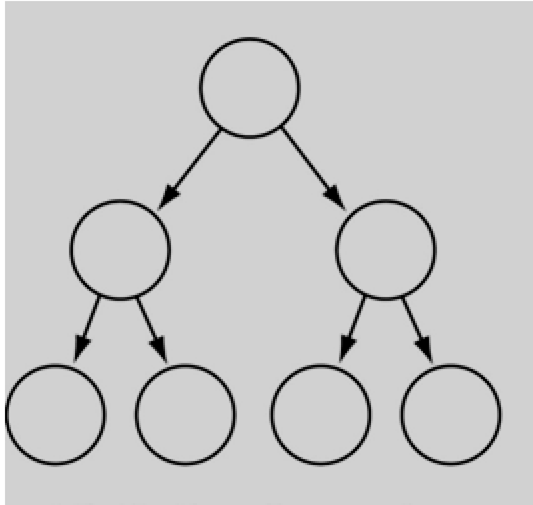
Heap

Outline

- Overview of Binary Tree
- Heap
- Operations
- Implementation
- Heap sort
- Example

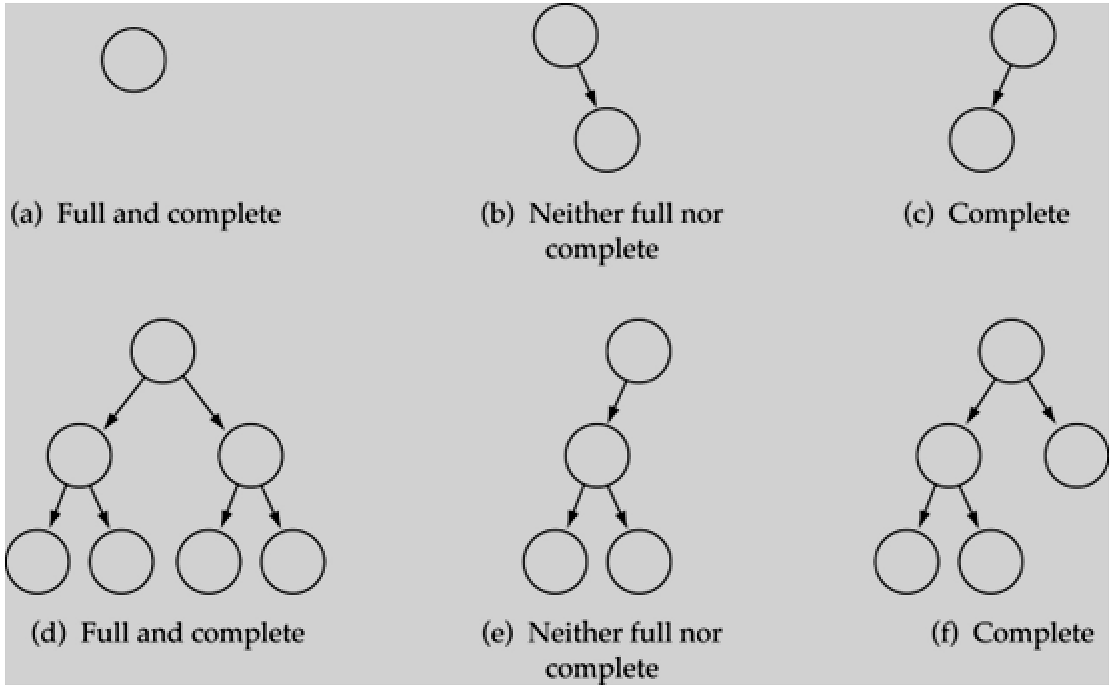
Outline

- Every non-leaf node has two children
- All the leaves are on the same level



Complete Binary Tree

- A binary tree that is either full or full through the next-to-last level
- The last level is full from left to right (i.e., leaves are as far to the left as possible)



(a) Full and complete

(b) Neither full nor complete

(c) Complete

(d) Full and complete

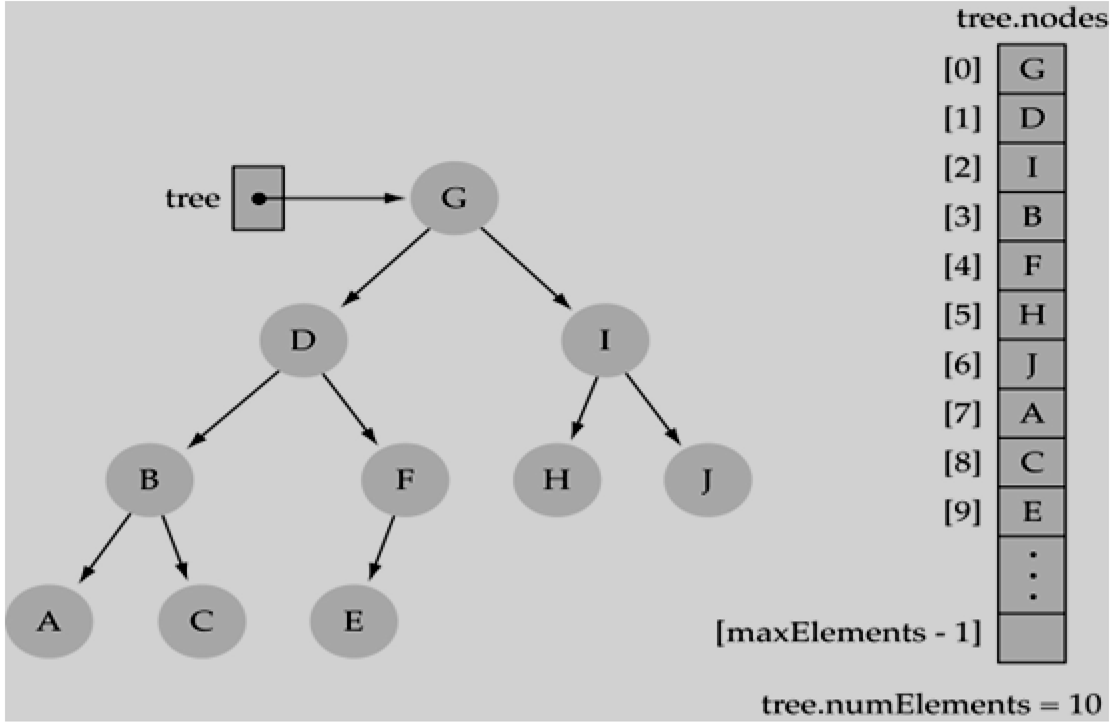
(e) Neither full nor complete

(f) Complete

Array-based representation of binary trees

- Memory space can be saved (no pointers are required)
- Preserve parent-child relationships by storing the tree elements in the array

(i) level by level, and (ii) left to right



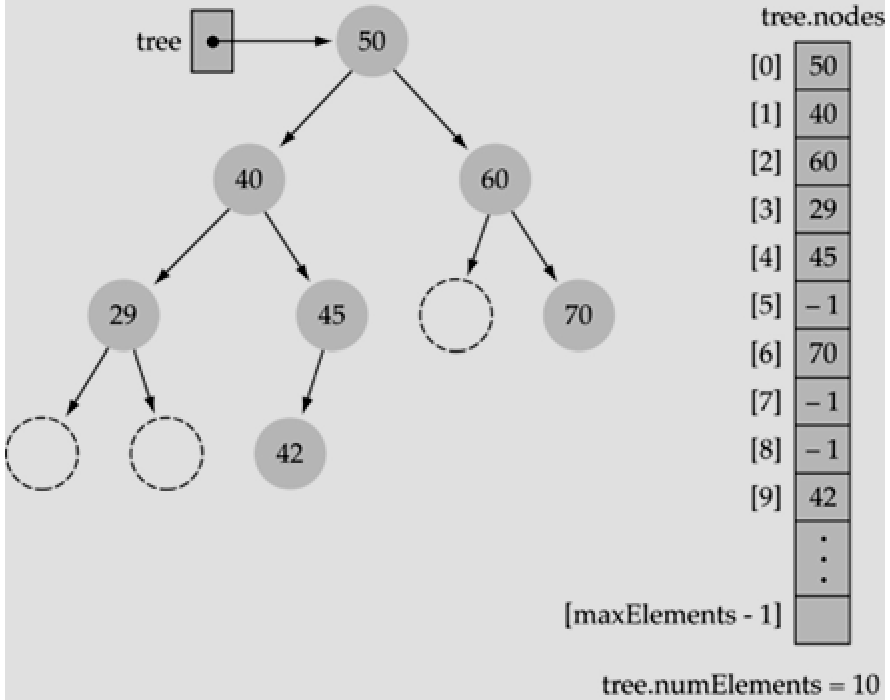
Array-based representation of binary trees

- Parent-child relationships:
  - left child of tree.nodes[index] = tree.nodes[2*index+1]
  - right child of tree.nodes[index] = tree.nodes[2*index+2]
  - parent node of tree.nodes[index] = tree.nodes[(index-1)/2]

    (int division-truncate)
- Leaf nodes:
  - tree.nodes[numElements/2] to tree.nodes[numElements - 1]

Array-based representation of binary trees

- Full or complete trees can be implemented easily using an array-based representation (elements occupy contiguous array slots)
- "Dummy nodes" are required for trees which are not full or complete

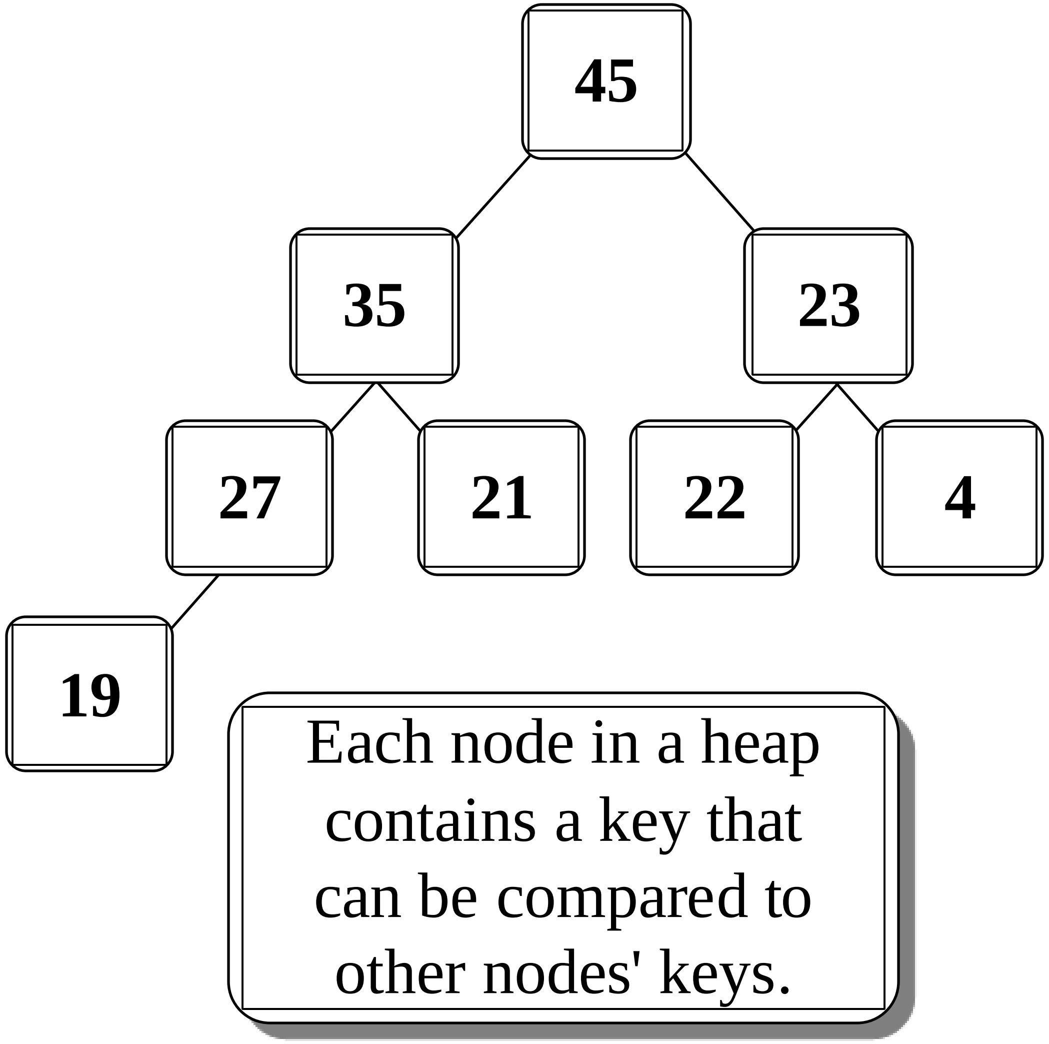


	tree.nodes
[0]	50
[1]	40
[2]	60
[3]	29
[4]	45
[5]	– 1
[6]	70
[7]	– 1
[8]	– 1
[9]	42
	⋮
[maxElements - 1]	

tree.numElements = 10

Heap

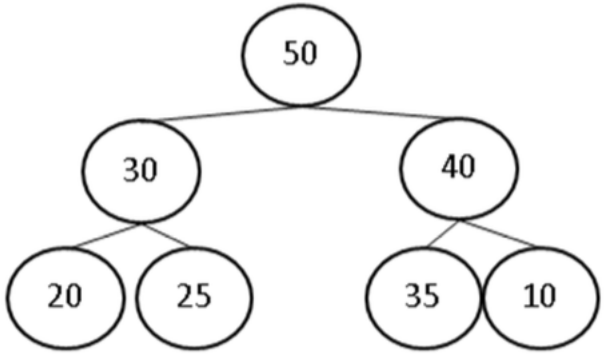
- It is a binary tree with the following properties:
  - Property 1: it is a complete binary tree
  - Property 2: the value stored at a node is greater or equal to the values stored at the children (Max heap)



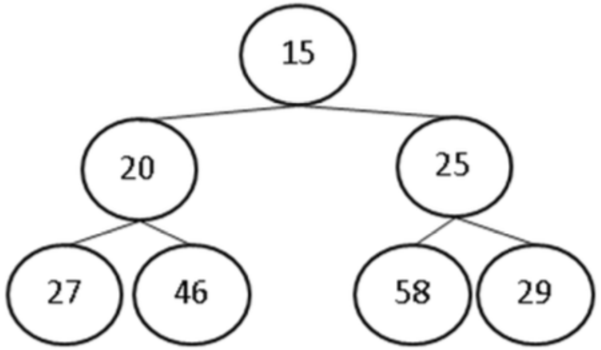
Each node in a heap contains a key that can be compared to other nodes' keys.

Heap

- There are 2 types of heap
  - Max heap: the value in the node is greater than its children.
  - Min heap: the value in the node is less than its children.

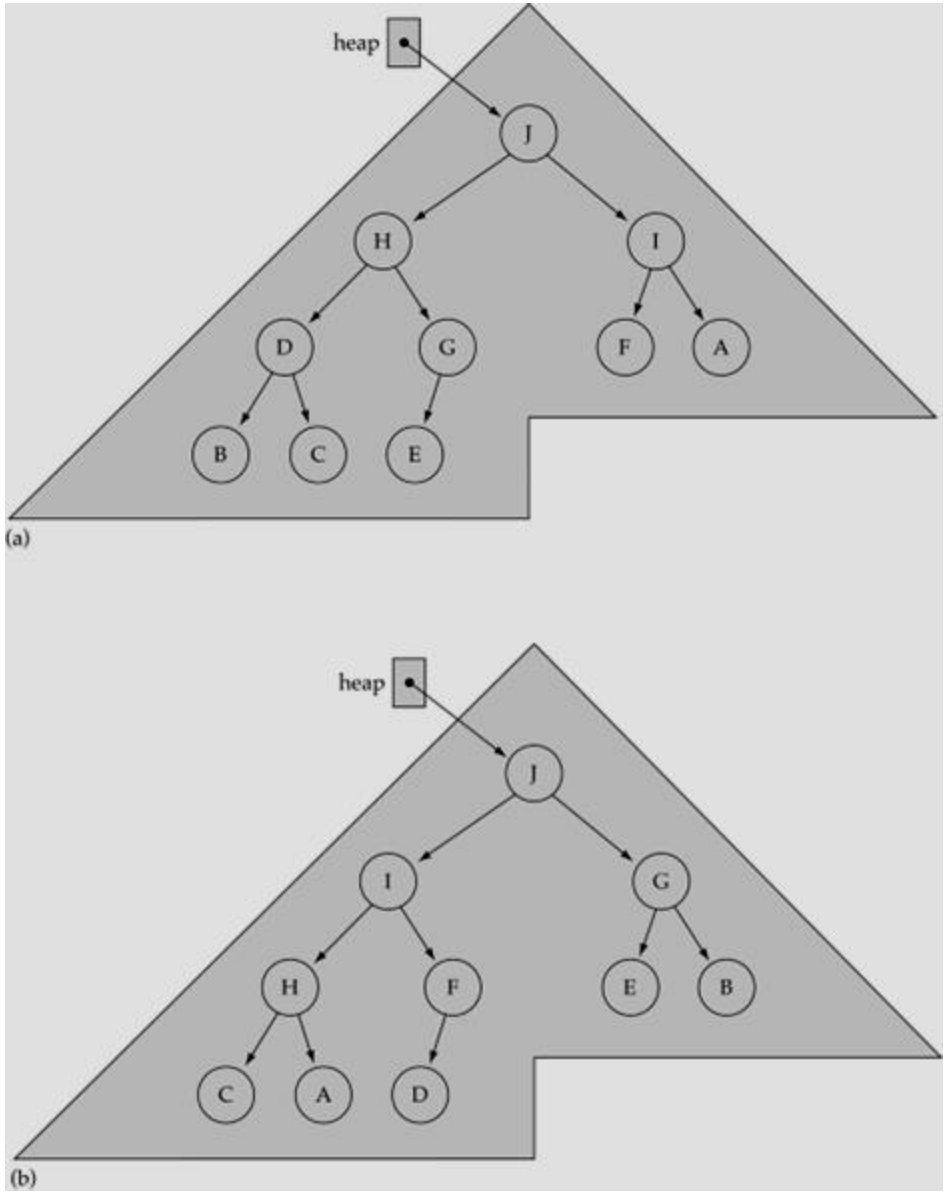


Max heap



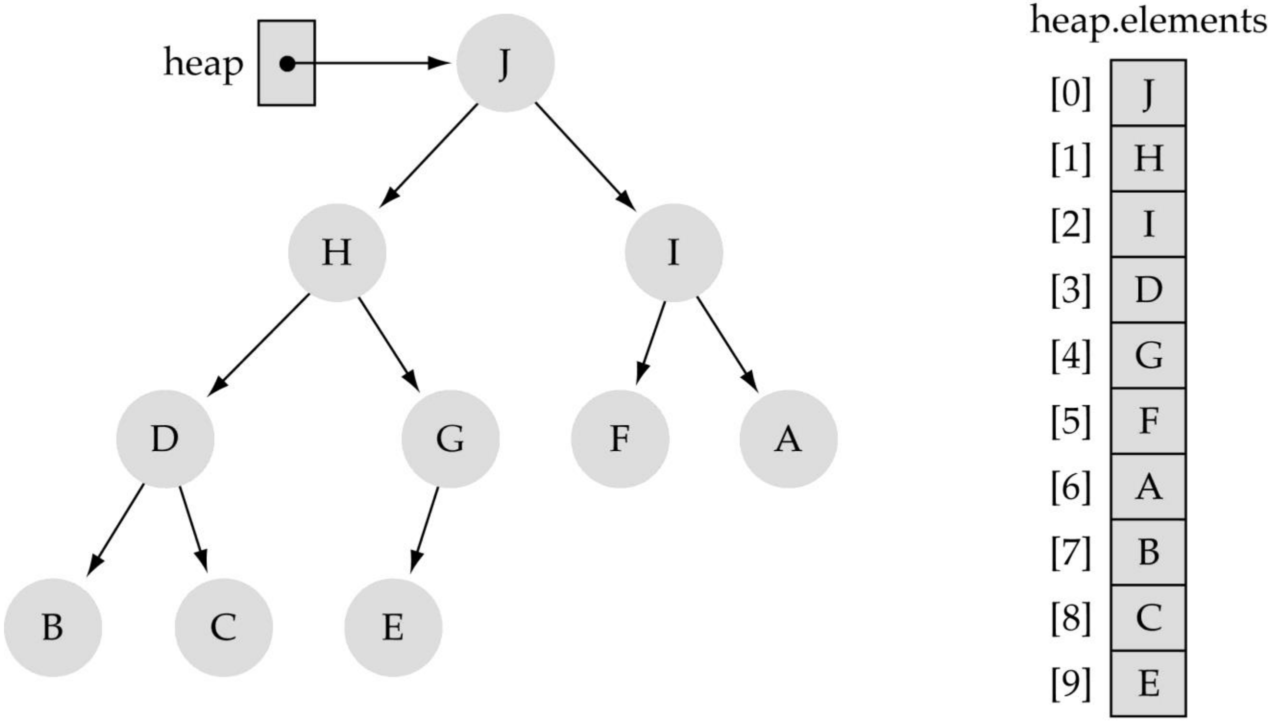
Min heap

Heap



Largest heap element

- From Property 2, the largest value of the heap is always stored at the root

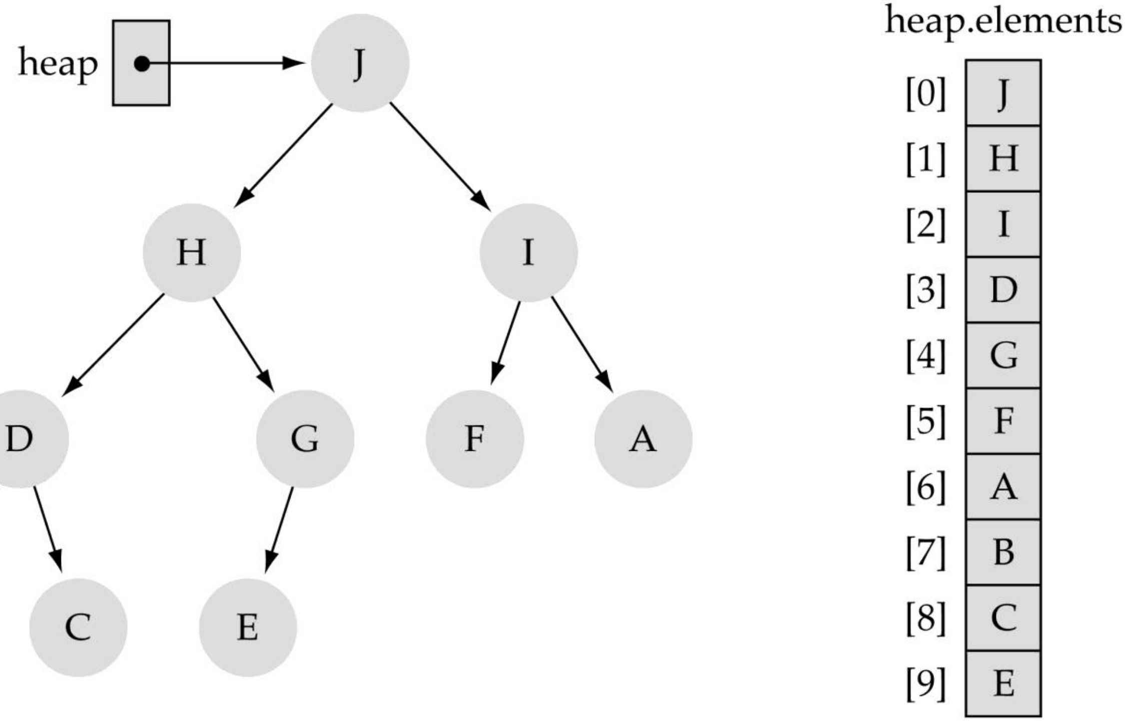


	heap.elements
[0]	J
[1]	H
[2]	I
[3]	D
[4]	G
[5]	F
[6]	A
[7]	B
[8]	C
[9]	E

*** This algorithm stores root at position 0.

Heap implementation using array representation

- A heap is a complete binary tree, so it is easy to be implemented using an array representation

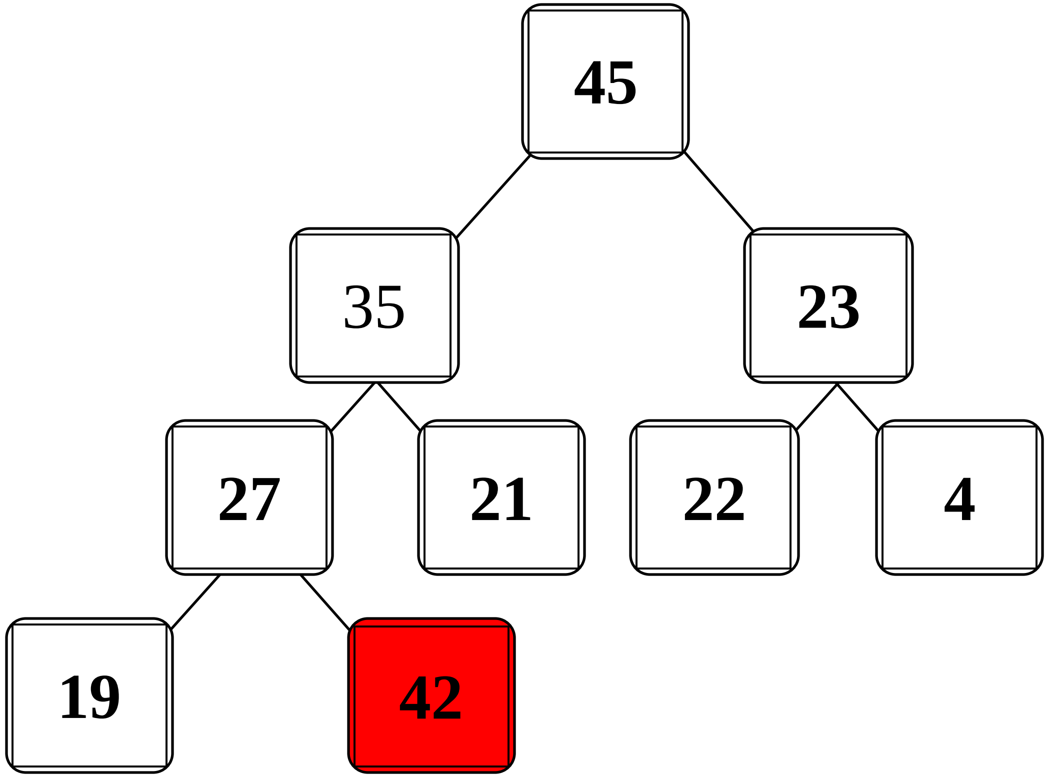


heap.elements

[0]	J
[1]	H
[2]	I
[3]	D
[4]	G
[5]	F
[6]	A
[7]	B
[8]	C
[9]	E

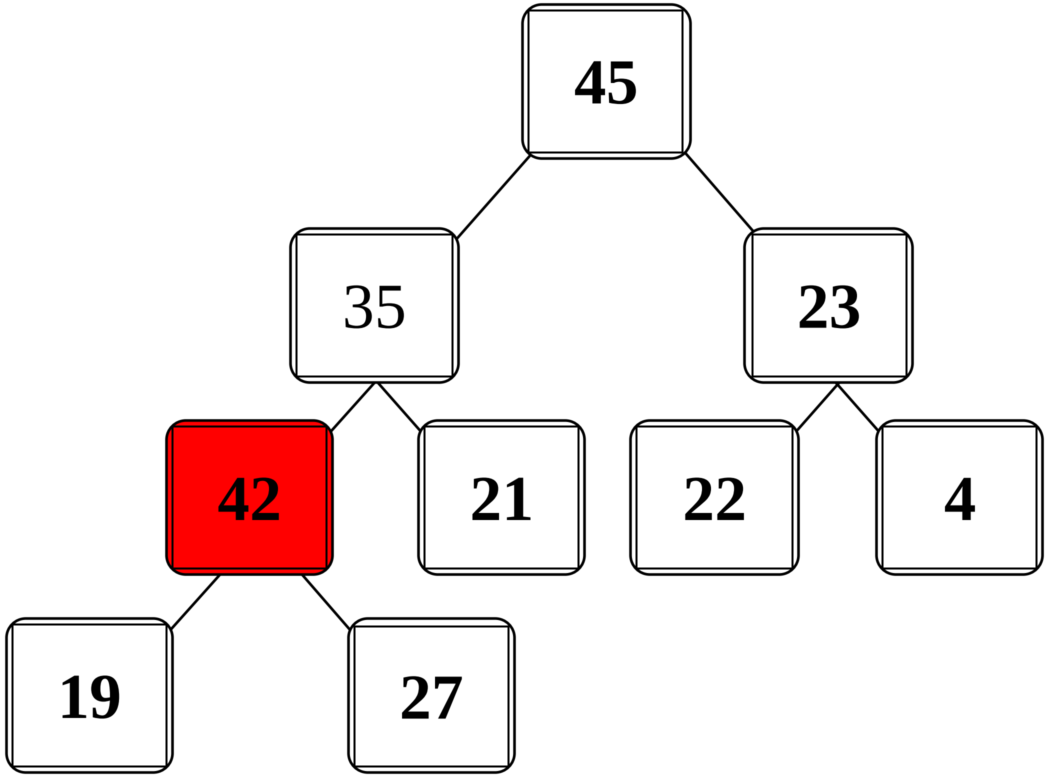
Add a node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



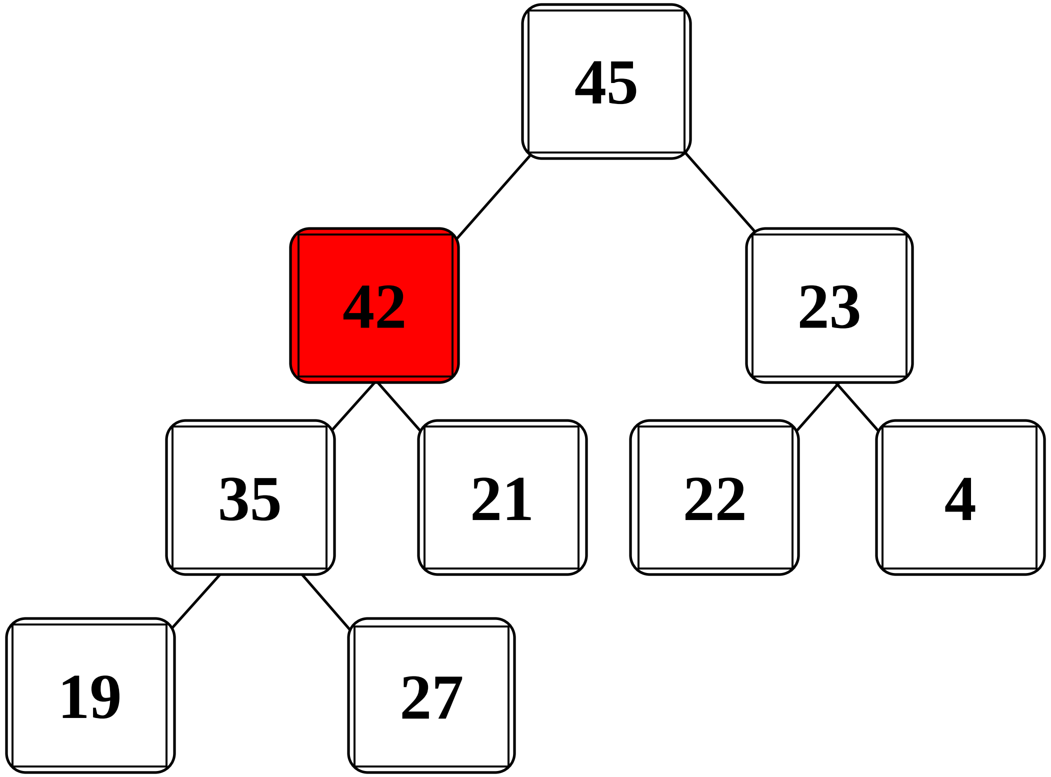
Add a node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



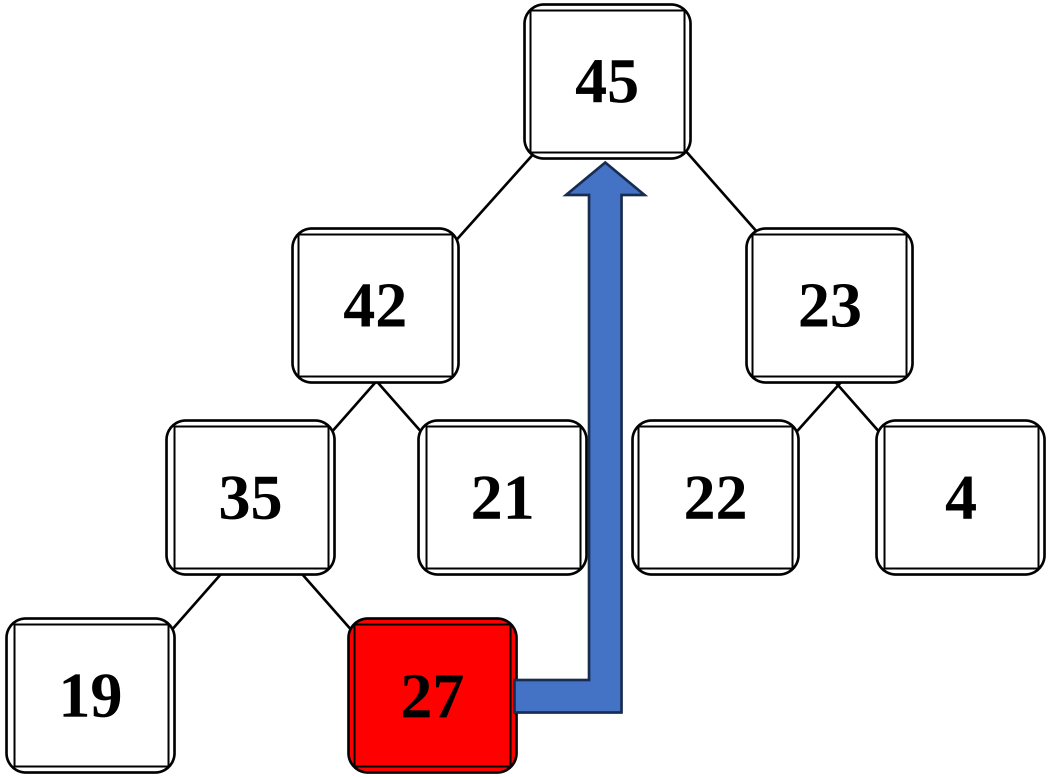
Add a node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



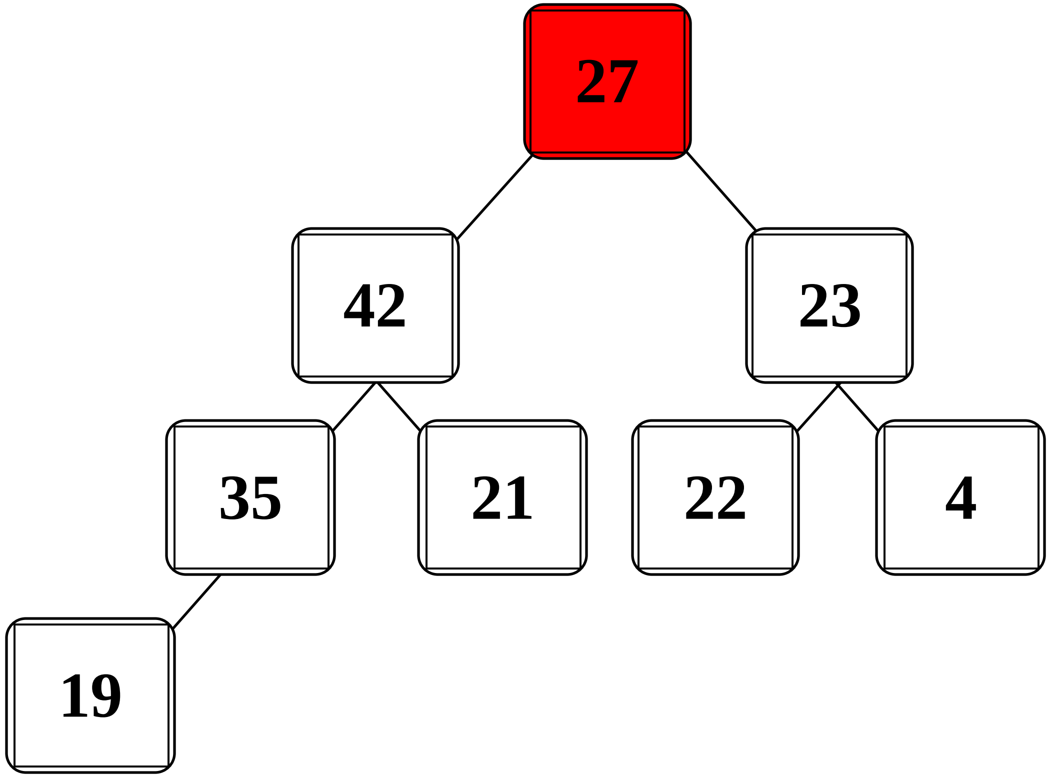
Removing the top of a Heap

- Move the last node onto the root.



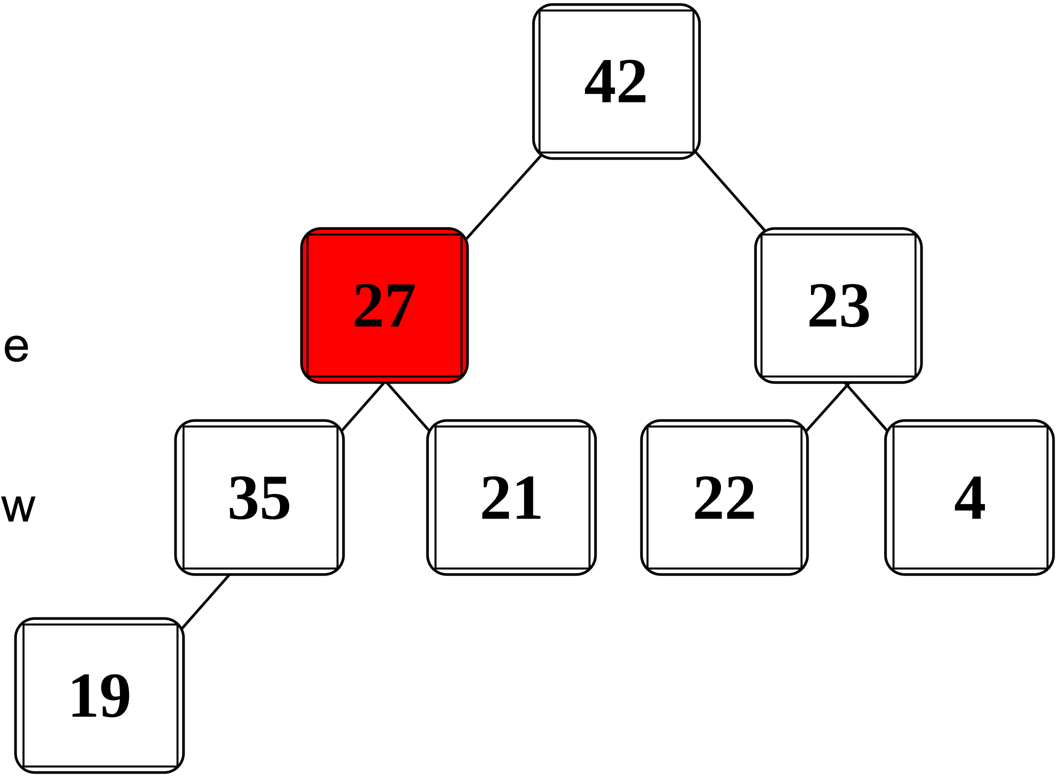
Removing the top of a Heap

- Move the last node onto the root.



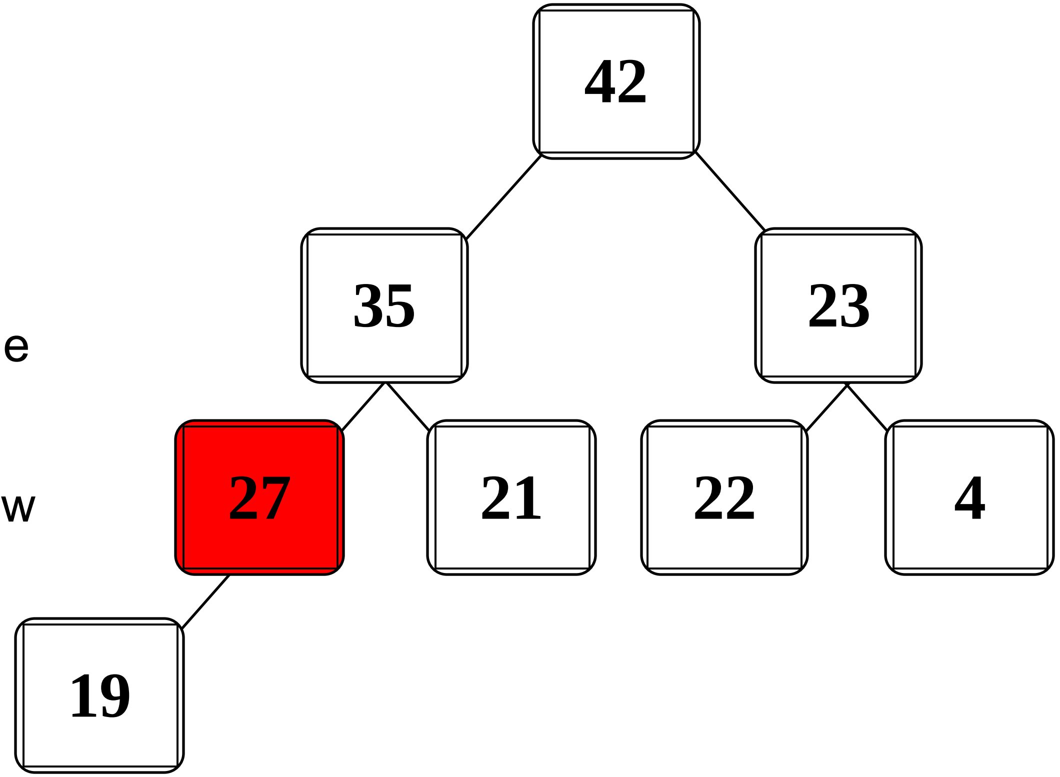
Removing the top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



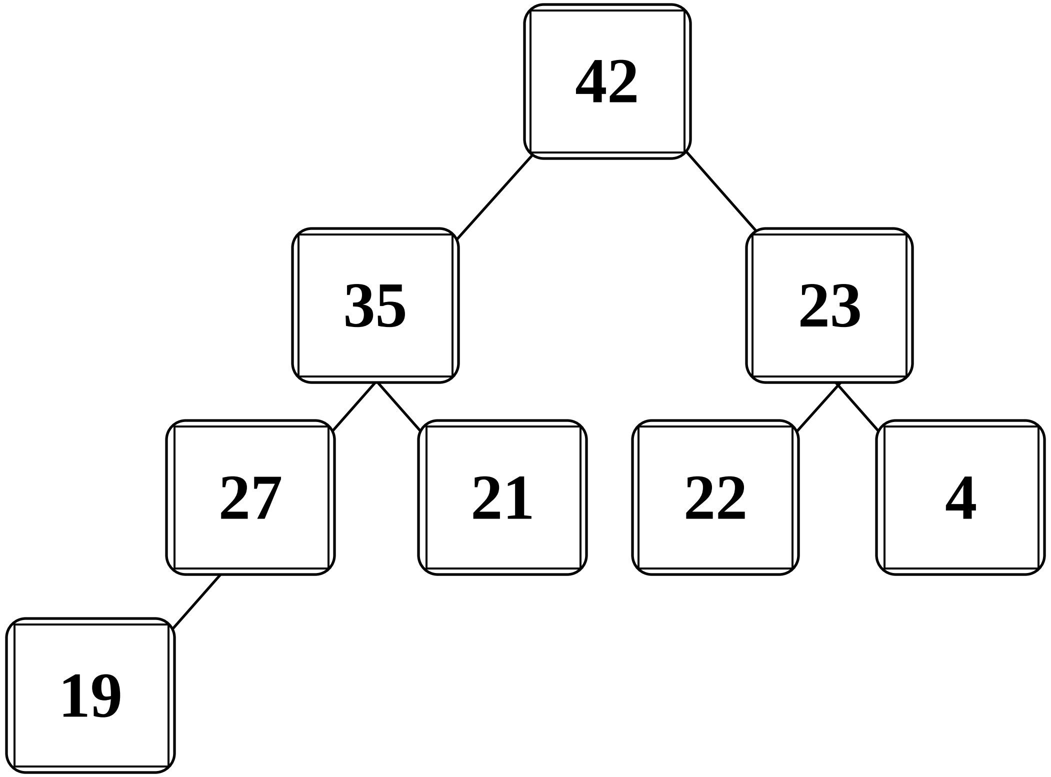
Removing the top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



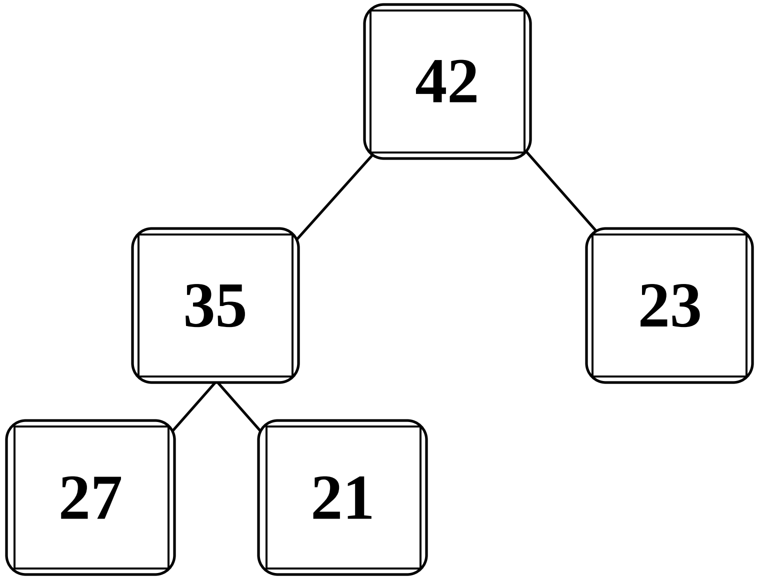
Removing the top of a Heap

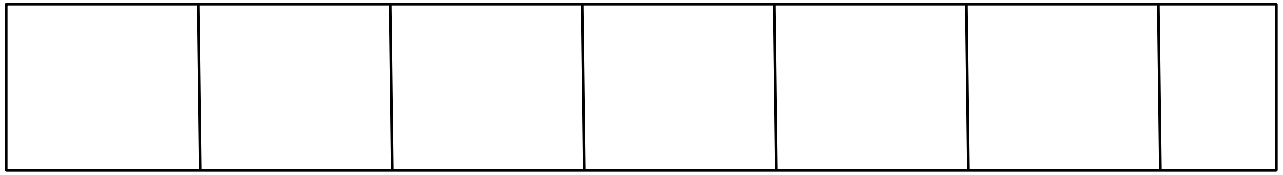
- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.
- The process of pushing the new node downward is called reheapification downward.



Implementation a Heap

- We will store the data from the nodes in a partially-filled array.

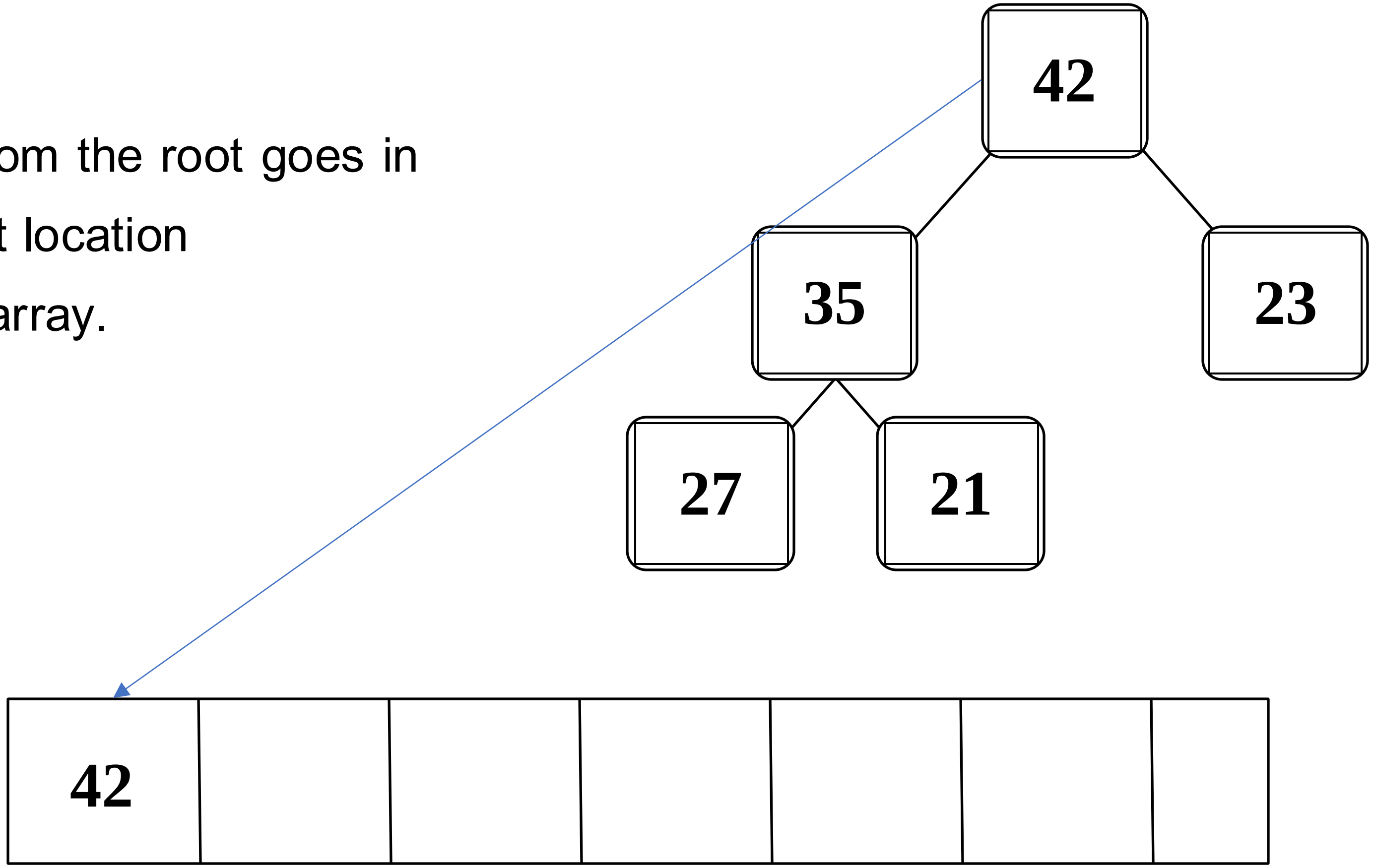




An array of data

Implementation a Heap

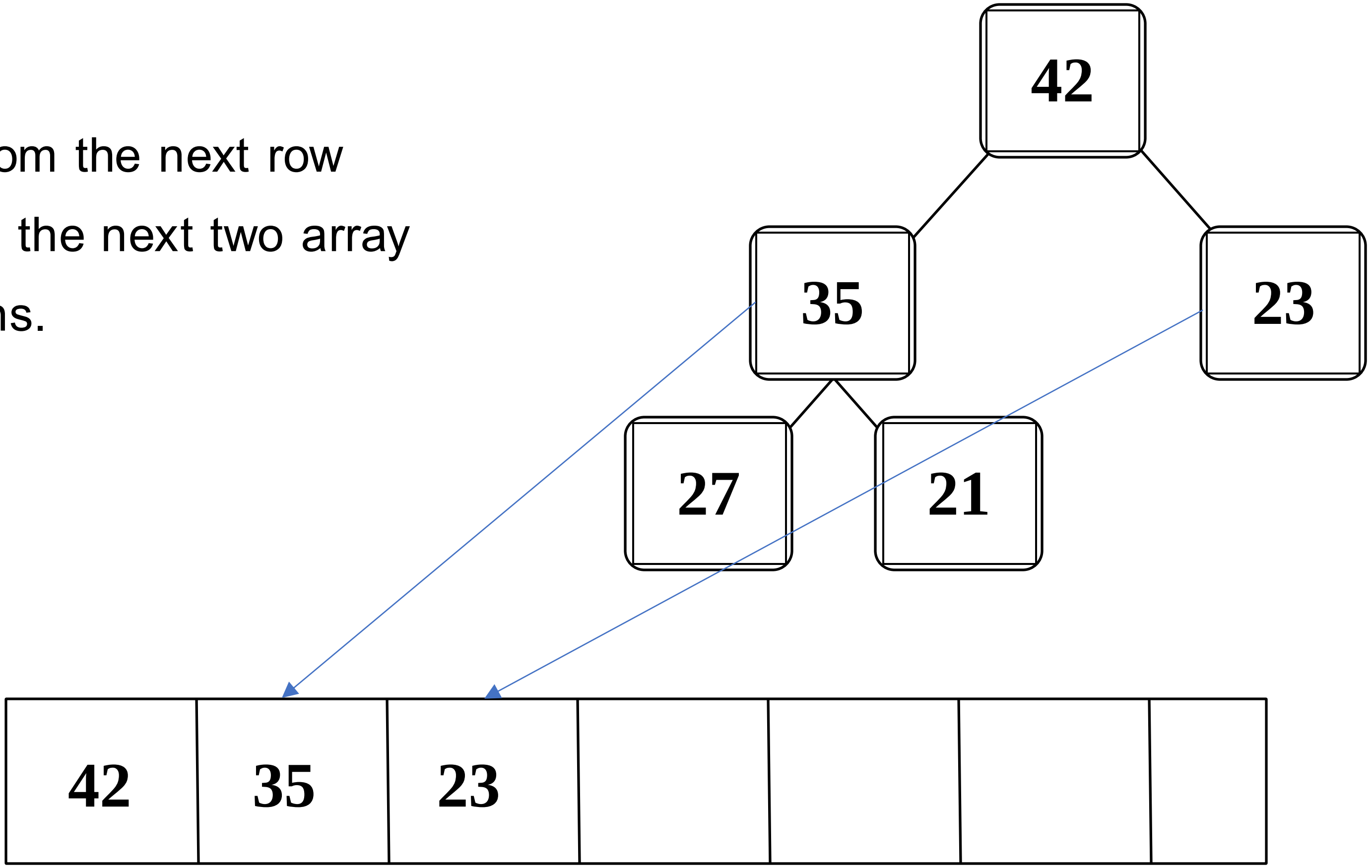
- Data from the root goes in the first location of the array.



An array of data

Implementation a Heap

- Data from the next row goes in the next two array locations.

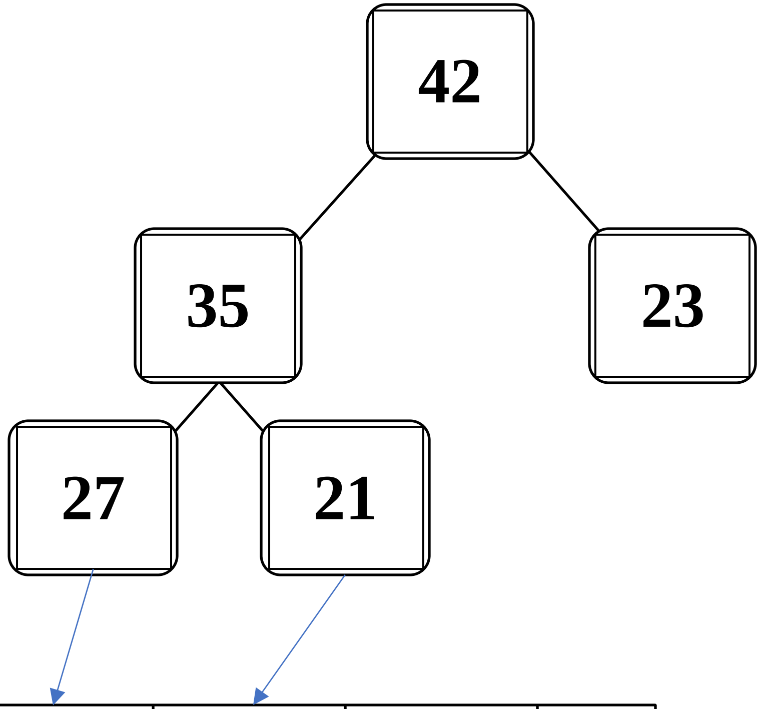


42	35	23				

An array of data

Implementation a Heap

- Data from the next row goes in the next two array locations.

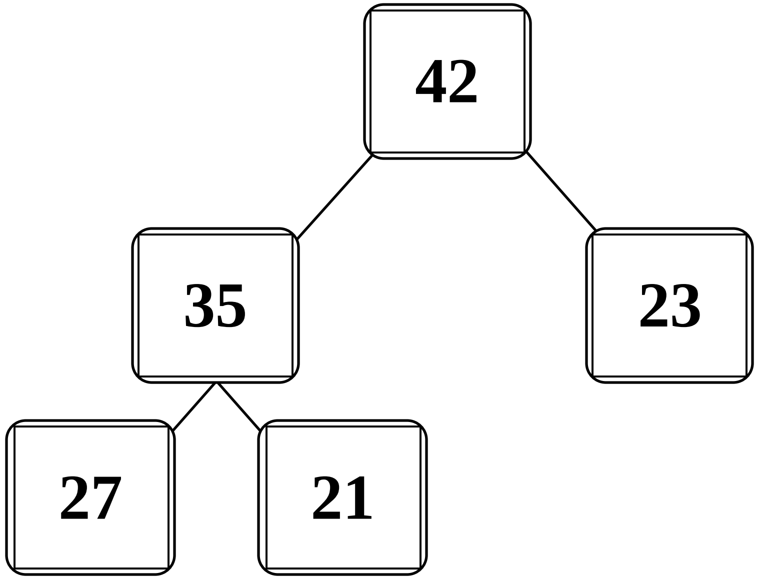


42	35	23	27	21		

An array of data

Implementation a Heap

- The links between the tree's nodes are not actually stored as pointers, or in any other way.
- The only way we "know" that "the array is a tree" is from the way we manipulate the data.

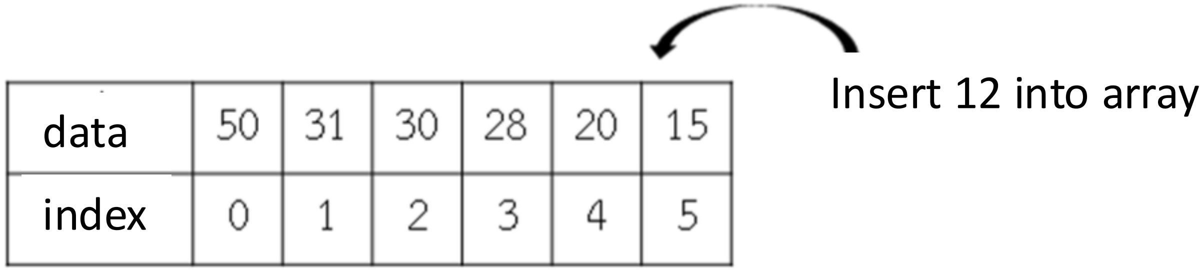


42	35	23	27	21		

An array of data

Priority Queues

- What is a priority queue?
  - It is a queue with each element being associated with a "priority"
  - From the elements in the queue, the one with the highest priority is dequeued first

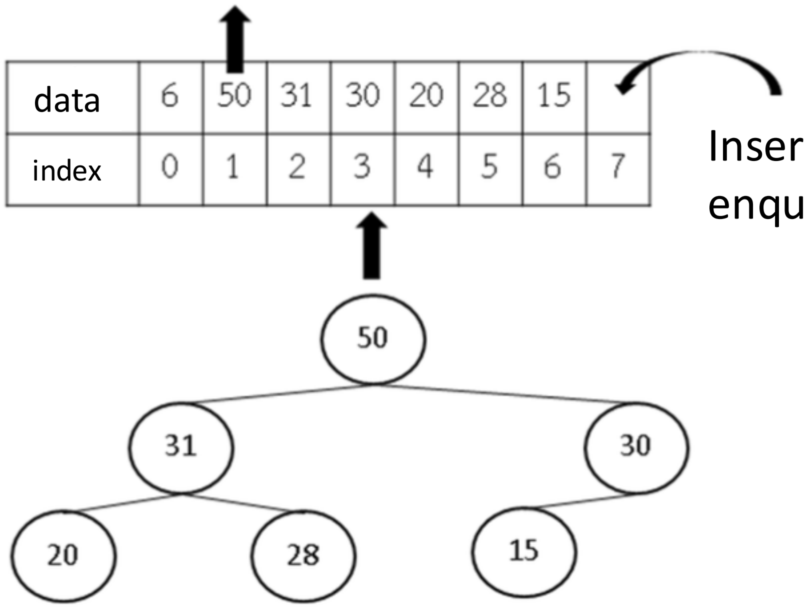


data	50	31	30	28	20	15
index	0	1	2	3	4	5

Insert 12 into array

Priority Queues

Remove data at root or dequeue



data	6	50	31	30	20	28	15	
index	0	1	2	3	4	5	6	7

Insert 12 to this queue or enqueue

***This algorithm stores root at position 1.

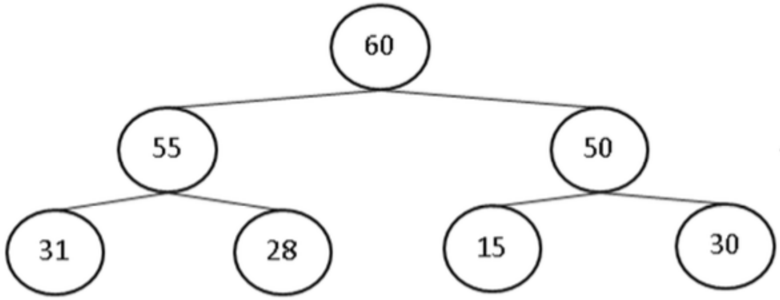
Heap Sort

- Heap sort is a comparison-based sorting technique based on Binary Heap data structure.
- It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning.
- Repeat the same process for the remaining elements.

Heap Sort

- The first step includes the creation of a heap by adjusting the elements of the array.
- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

Heap Sort

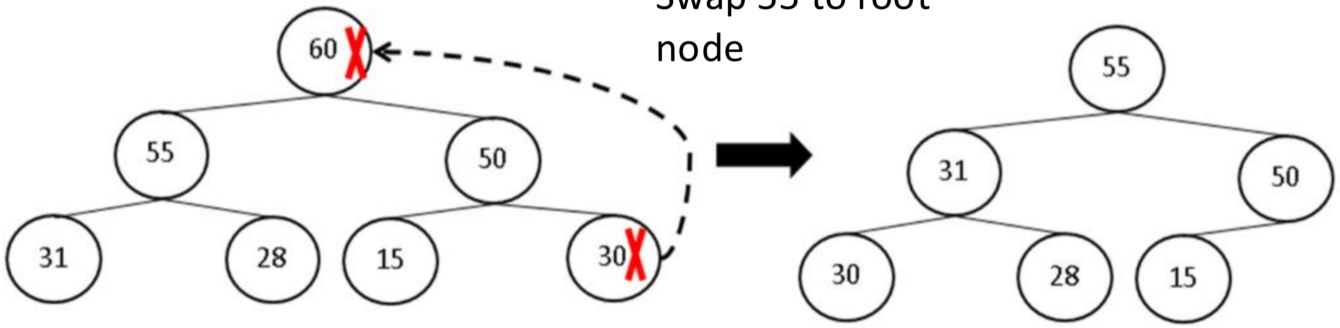


Max heap

data	7	60	55	50	31	28	15	30
index	0	1	2	3	4	5	6	7

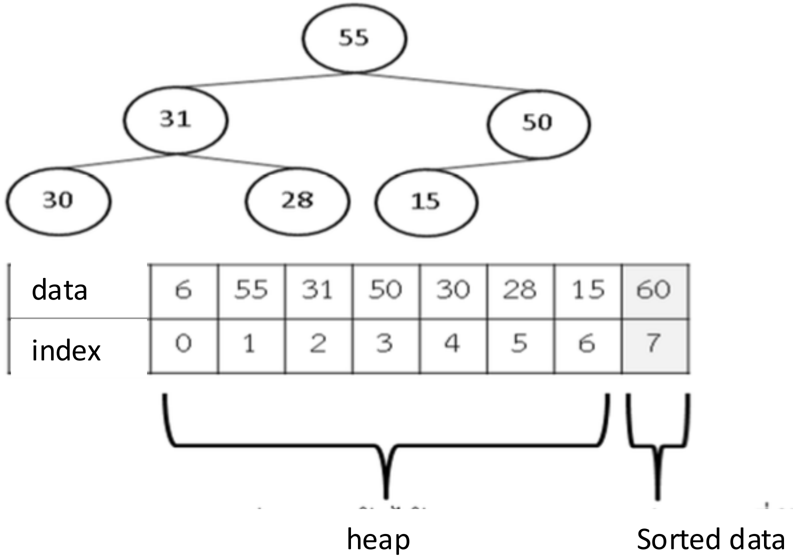
Heap Sort

Dequeue (delete at root)



Swap 55 to root node

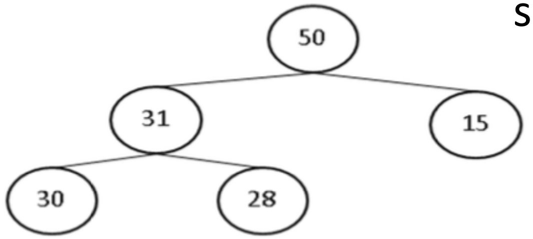
Heap Sort



data	6	55	31	50	30	28	15	60
index	0	1	2	3	4	5	6	7

Heap Sort

run until data are sorted



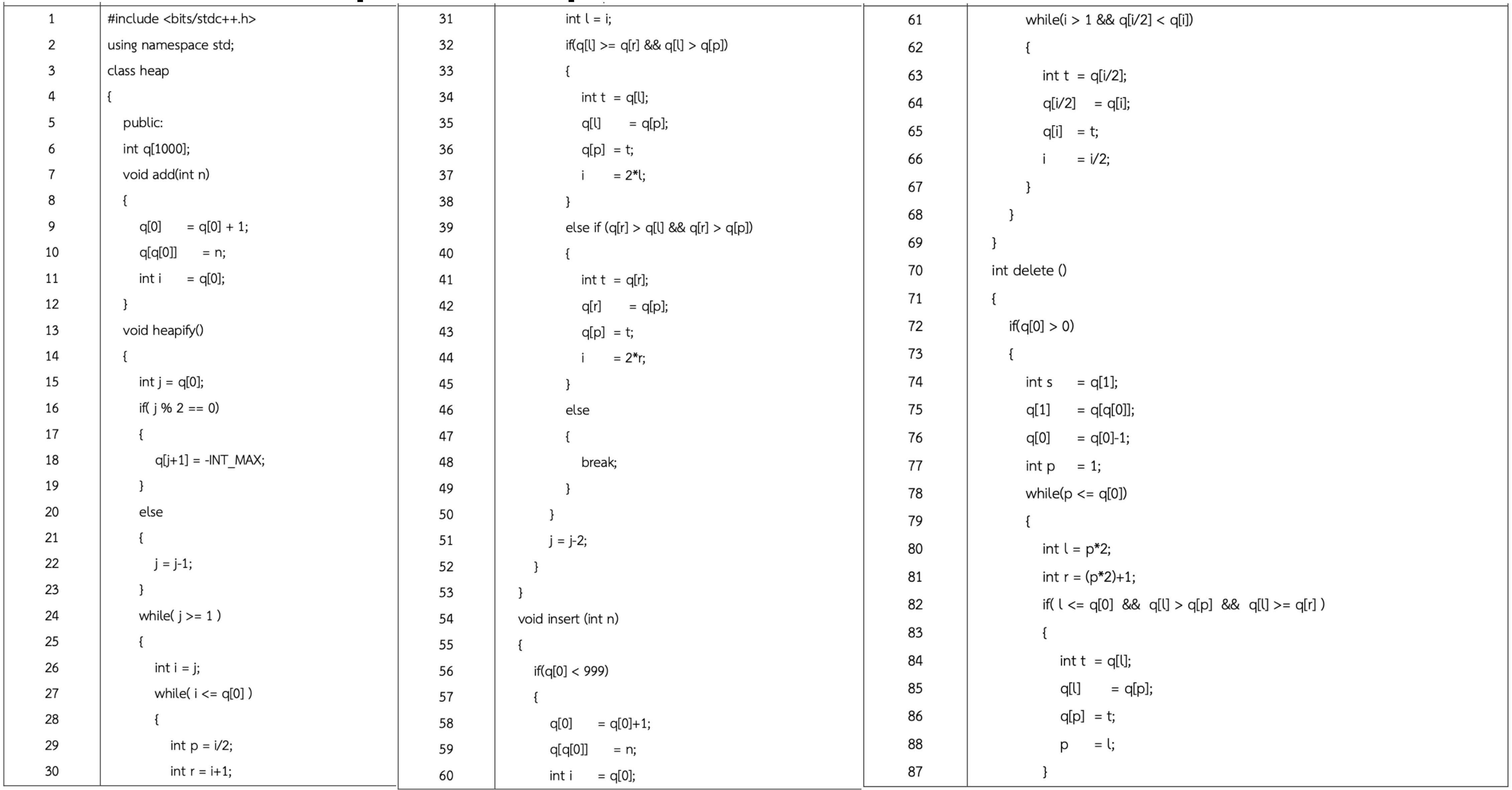
data	5	50	31	15	30	28	55	60
index	0	1	2	3	4	5	6	7

heap

Sorted data

data	0	5	15	30	31	50	55	60
index	0	1	2	3	4	5	6	7

Max heap example



```
#include <bits/stdc++.h>
using namespace std;
class heap
{
    public:
    int q[1000];
    void add(int n)
    {
        q[0]      = q[0] + 1;
        q[q[0]]      = n;
        int i      = q[0];
    }
    void heapify()
    {
        int j = q[0];
        if( j % 2 == 0)
        {
            q[j+1] = -INT_MAX;
        }
        else
        {
            j = j-1;
        }
        while( j >= 1 )
        {
            int i = j;
            while( i <= q[0] )
            {
                int p = i/2;
                int r = i+1;
                int l = i;
                if(q[l] >= q[r] && q[l] > q[p])
                {
                    int t  = q[l];
                    q[l]       = q[p];
                    q[p]  = t;
                    i       = 2*l;
                }
                else if (q[r] > q[l] && q[r] > q[p])
                {
                    int t  = q[r];
                    q[r]       = q[p];
                    q[p]  = t;
                    i       = 2*r;
                }
                else
                {
                    break;
                }
            }
            j = j-2;
        }
    }
    void insert (int n)
    {
        if(q[0] < 999)
        {
            q[0]      = q[0]+1;
            q[q[0]]      = n;
            int i      = q[0];
            while(i > 1 && q[i/2] < q[i])
            {
                int t  = q[i/2];
                q[i/2]    = q[i];
                q[i]   = t;
                i       = i/2;
            }
        }
    }
    int delete ()
    {
        if(q[0] > 0)
        {
            int s      = q[1];
            q[1]      = q[q[0]];
            q[0]      = q[0]-1;
            int p      = 1;
            while(p <= q[0])
            {
                int l = p*2;
                int r = (p*2)+1;
                if( l <= q[0]  &&  q[l] > q[p]  &&  q[l] >= q[r] )
                {
                    int t  = q[l];
                    q[l]       = q[p];
                    q[p]  = t;
                    p      = l;
                }
```

Max heap example

```
                else if ( r <= q[0] && q[r] > q[p] && q[r] > q[l])
                {
                    int t  = q[r];
                    q[r]     = q[p];
                    q[p]     = t;
                    p     = r;
                }
                else
                {
                    break;
                }
            }
            return s;
        }
        else
        {
            return NULL;
        }
    }
    void print()
    {
        for(int i=1 ; i <= q[0] ; i++)
        {
            cout<<q[i]<<" ";
        }
        cout<<endl;
    }
};
```

```
int main()
{
    heap h;
    h.add(55); h.add(14); h.add(50); h.add(20); h.add(5);
    h.add(15); h.add(30); h.add(31); h.add(60); h.add(28);
    h.print();
    h.heapify();                h.print();
    cout<<h.delete()<<" : ";    h.print();
    cout<<h.delete()<<" : ";    h.print();
    cout<<h.delete()<<" : ";    h.print();
    cout<<h.delete()<<" : ";    h.print();
    cout<<h.delete()<<" : ";    h.print();
    cout<<h.delete()<<" : ";    h.print();
    cout<<h.delete()<<" : ";    h.print();
    cout<<h.delete()<<" : ";    h.print();
    cout<<h.delete()<<" : ";    h.print();
    cout<<h.delete()<<" : ";    h.print();
    cout<<h.delete()<<" : ";    h.print();
    h.insert (15);              h.print();
    h.insert(13);               h.print();
    h.insert(9);                h.print();
    h.insert(20);               h.print();
    h.insert(8);                h.print();
    h.insert(11);               h.print();
    h.insert(30);               h.print();
    h.insert(2);                h.print();
}
```

Reference

Allen, W. M. (2007). *Data structures and algorithm analysis in C++*. Pearson Education India.

Nell B. Dale. (2003). *C++ plus data structures*. Jones & Bartlett Learning.

เฉียบวุฒิ รัตนวิลัยสกุล. (2023). โครงสร้างข้อมูล. มหาวิทยาลัยเทคโนโลยีพระจอมเกล้าพระนครเหนือ

https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/