1. **Structure Property**
   A heap is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right.

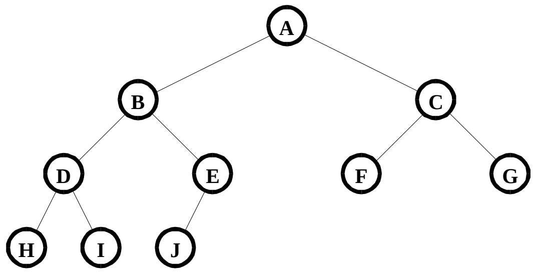
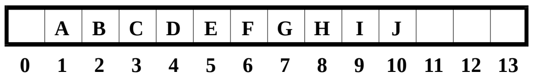


Figure 1. A complete binary tree.



	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Figure 2. Array implementation of the tree in Figure 1.

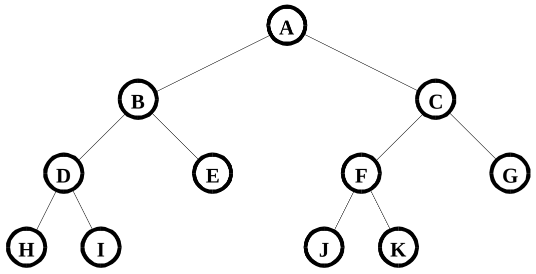


Figure 3. Binary tree that is *not* filled from left to right.

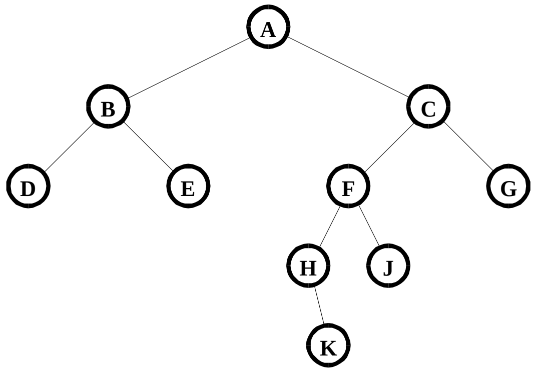


Figure 4. Incomplete binary tree

2. **Number of nodes and Height**
   The number of nodes, *N*, in a binary tree of height, *h* is: $2^h \le N \le 2^{h+1} - 1$.
   The height of a binary tree is: $h = \lfloor \log_2 N \rfloor$.
3. **Position of children**
   Assume *i* is the index of a node in a binary tree. The left child is in position 2*i*. The right child is in position 2*i*+1.
   Example:
   The left child of node B is D. Node B is in position 2. Node D should be in position 2(2)=4. Note Node D is in position 4. Node E is the right child of node B. Node E should be in position 2(2)+1=5 and it is.
4. **Position of parent**
   If *i* is the position of any node is the tree except the root, the position of the parent of *i* is $\lfloor i/2 \rfloor$.
   Example:
   Node G is in position 7. Calculating the position of the parent of G, $\lfloor 7/2 \rfloor = 3$.
   Note C, the parent of G is, indeed, in position 3.
5. **Heap Order Property**
   For every node *X*, the value in the parent of X is smaller than or equal to the value in *X*. The foregoing rule applies to every node except the root that has no parent.

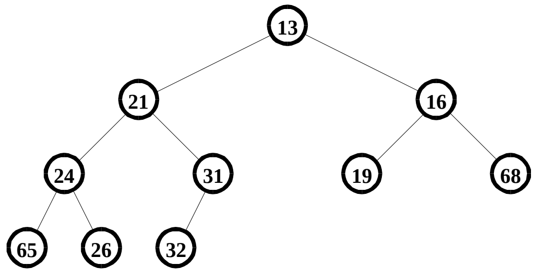


Figure 5. Complete tree that is also a heap

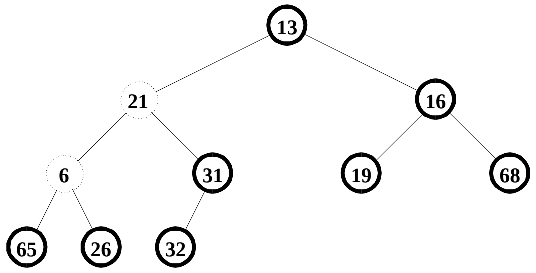


Figure 6. Complete tree that is *not* a heap

Figure 7. Definition of **class** *Heap*.

```
class Heap {
    int size;
    int count;
    int* H;
    void Graph
        (int i
        ,int level
        ,ostream& o
        );
public:
    class HeapFullException{};
    class HeapEmptyException{};
    Heap(int sz=100);
     ~Heap();
    bool IsFull(void);
    bool IsEmpty(void);
    void Insert(int v);
    int Remove(void);
    void Print
        (ostream& o
        ,char* title
        );
    void Scan(istream& i);
    void Sort(void);
    void Graph(ostream& o);
};
```

Members of **class** *Heap*.

1. Member *size* contains the number of integer locations available for use by the heap.
2. Member *count* enumerates the number of integer locations in actual use by the heap.
3. Member *H* points to an integer array used to store elements of the heap.
4. Member function *Graph* prints elements of the heap indented according their depth. An inorder traversal is used. Elements are printed on separate line.
5. Constructor *Heap* initializes data members size and *count*. Storage for the heap is allocated and assigned to member *H*. The first element of member *H* is a sentinel and assigned the minimum integer.
6. Destructor *~Heap* returns dynamically allocated storage.
7. Member function *IsFull* determines if the heap is full.
8. Member function *IsEmpty* determines if the heap is empty.
9. Member function *Insert* inserts an integer into the heap.
10. Member function *Remove* removes and returns the element having the smallest value.
11. Member function *Print* prints the elements of the heap in index order.
12. Member function *Scan* reads a file containing integers separated by white space. Integers are inserted into the heap.
13. Member function *sort* sorts the heap in ascending order. The result is also a heap.
14. Member function *Graph* prints the elements of the heap according to their depth from the root. Elements are printed on separate lines. An inorder traversal is used to print elements.

Member function *Insert*

1. Create a hole in the next available position in the heap.
2. Insert the new value in the hole
3. Percolate the new element up to its appropriate location in the heap
4. Percolation is terminated by the existence of a sentinel having a value smaller than any in the heap in position zero (0).

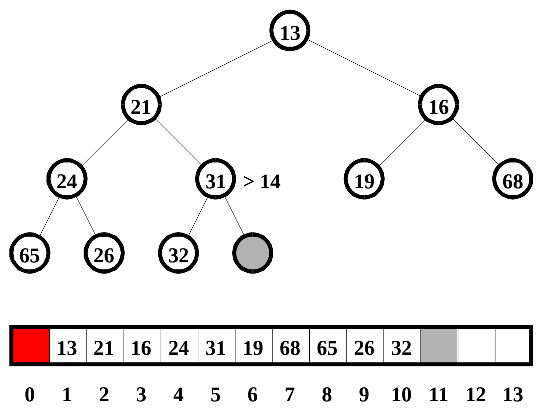


Figure 8. Inserting new value 14, step 1

1. Create a hole at the end of the heap in position 11 in Figure 8.
2. Compare the value of the parent, in position 5, with the key. Refer to Figure 8.
3. If the value of the parent (31) is greater than the key, copy the value of parent to the child – the newly created hole. Refer to Figure 9.

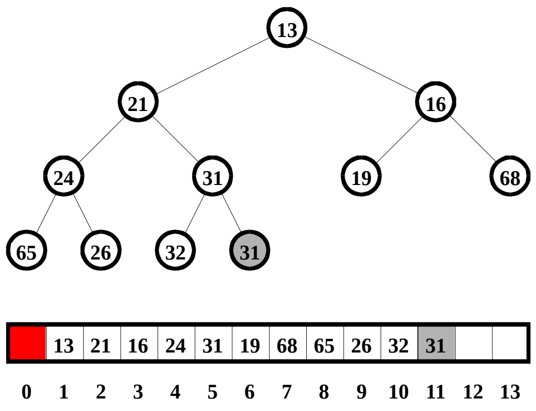


Figure 9. Inserting new value 14, step 2

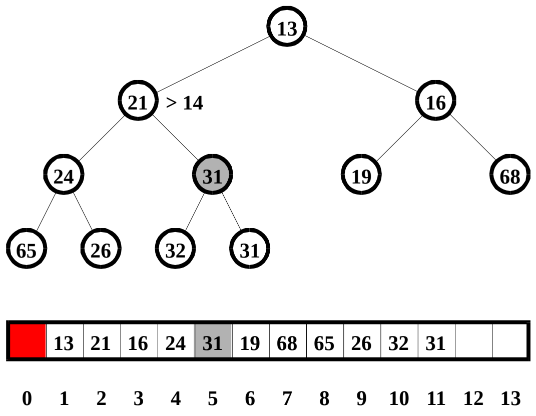


Figure 10. Inserting new value 14, step 3

Compare the value (21) of the parent in position (2) with the key. If the parent is greater than the key, and it is, copy the value of the parent to the child in position 5. Figure 10 illustrates the comparison. The child is shaded in Figure 10. The value of the parent is copied to the child in Figure 11.

Figure 11. Inserting new value 14, step 4

Figure 12. Inserting new value 14, step 5

Compare the value of the parent (13) in position 1 with the key. If the parent is greater than the key, copy the parent to the child. The parent is not, however, greater than the key. Instead, assign the value of the key to the child. The child is shaded in Figures 12 and 13.

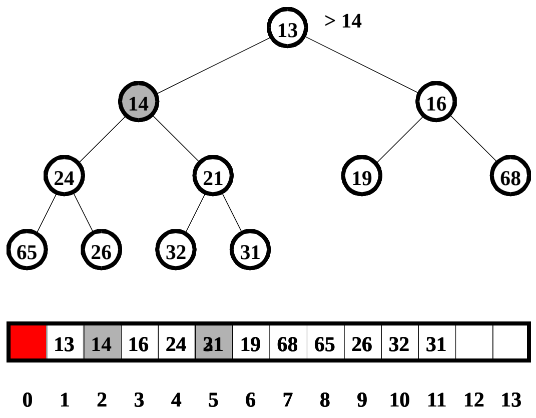


Figure 13. Inserting new value 14, step 6

Member function *Remove*

1. Remove minimum element at the root and leave a hole at the root.
2. Decrement the number of elements in the heap
3. Copy the smaller of the two children into the hole and move the hole to the position of the child that was copied.
4. Move to the smaller child

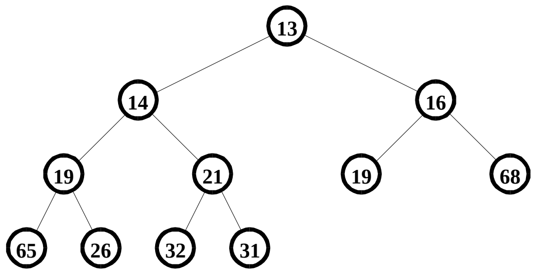


Figure 14. Remove, step 1

Heap member function remove deletes and returns the minimum element at the root of the heap. The value thirteen (13) is at the top of the heap in Figure 14.

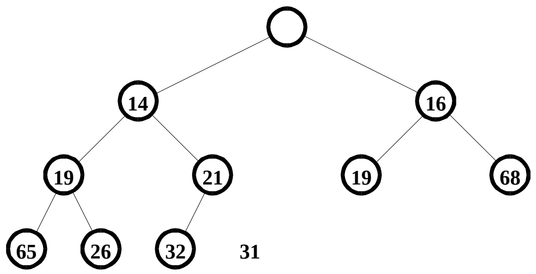
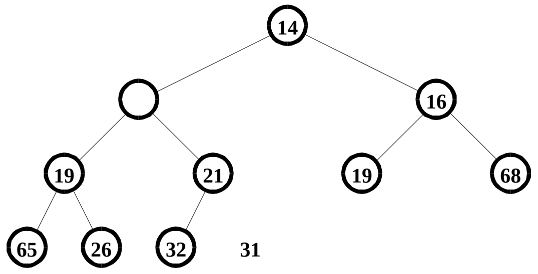


Figure 15. Remove, step 2

A hole is created at the top of the heap where the minimum element has been removed in Figure 15. The size of the heap has been diminished by one value. Diminishing the size of the heap is illustrated by the absence of the circle around the last element (31).

Figure 16. Remove, step 3

The smaller of the two children is copied to the hole created in the previous step. A new hole is created where the child was.



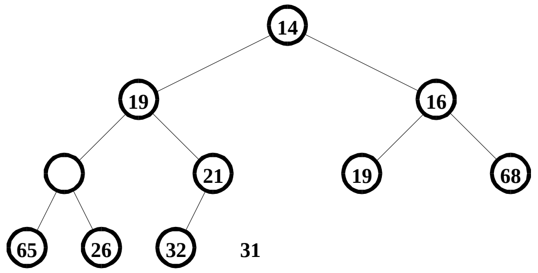


Figure 17. Remove, step 4

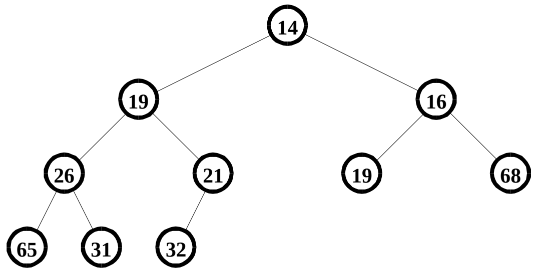


Figure 18. Remove, step 5

```
Heap::Heap(int sz):count(0),size(sz)
{     H=new int[size];
      H[0]=INT_MIN;
}
```

Figure 16. class *Heap* constructor.

```
Heap::~Heap() { if (H) delete[] H; }
```

Figure 17. class *Heap* destructor.

```
void Heap::Insert(int v)
{     if (IsFull()) throw HeapFullException();
      int a=++count;
      while (H[a/2]>v) {
            H[a]=H[a/2];
            a/=2;
      }
      H[a]=v;
```

Figure 18. class *Heap* member function *Insert*..

```
int Heap::Remove(void)
{     if (IsEmpty()) throw HeapEmptyException();
      int min=H[1];
      int last=H[count--];
      int a,child;
      //-----------------------------------------------------------------------------------------------------
      //The correction a*2<=count (formerly a*2<count) is due to the indefatigable testing
      //of Mr. Glenn Billings.
      //-----------------------------------------------------------------------------------------------------
      for (a=1;a*2<=count;a=child) {
            child=a*2;
            if ((child!=count)&&(H[child+1]<H[child])) child++;
            if (last>H[child]) H[a]=H[child]; else break;
      }
      H[a]=last;
      return min;
}
```

Figure 19. class *Heap* member function *Remove*.