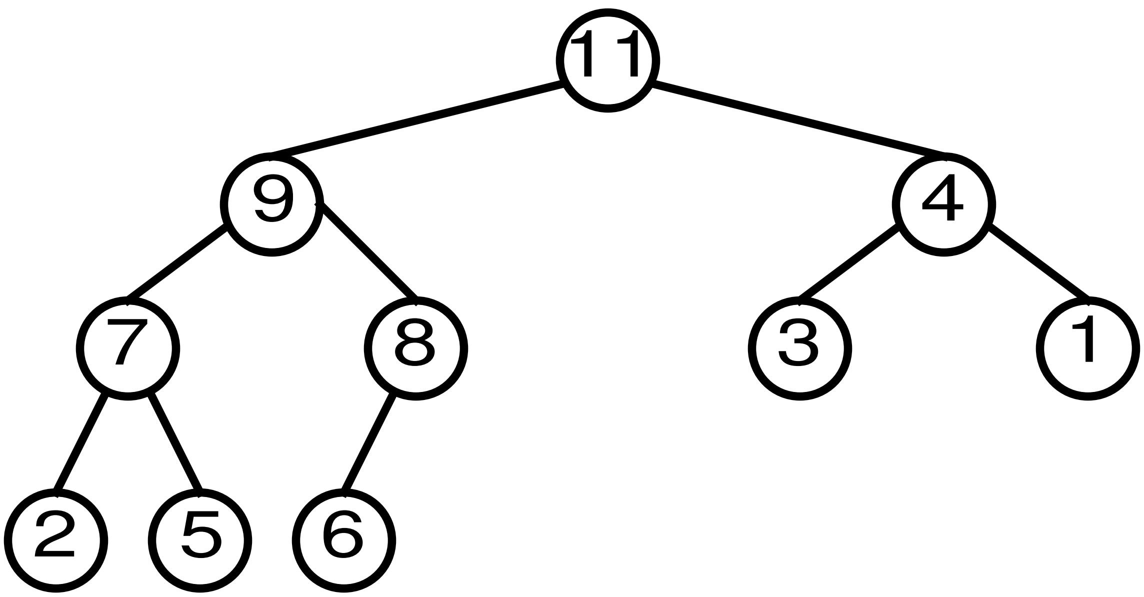
Heaps

Definition. A *max(min)-heap* is a complete binary tree in which the value stored at any node is greater (smaller) than the values stored at any of its children.



11	9	4	7	8	3	1	2	5	6	

Notes.

1. Because a heap is a complete binary tree, it will **almost always implemented as an array**.
2. Due to the partial order property of a max(min)-tree **the value stored at the root will always contain the maximum** (minimum) of all values in the heap.
3. In the followings we **will only consider max-heaps** which we shall simply call heaps.

Building a heap

1. **Inserting elements** one at a time - must maintain the heap properties

- cost: $\theta(n \lg n)$

2. **Heapifying** an array containing the values to be stored in the heap

- approach starting from the deepest level, sift the root of a subtree down to its position

- Cost: $\theta(n)$

Example. Build a heap from:

2, 3, 1, 10, 9, 7, 8, 4, 6, 5

The heap ADT

Type: a set of heaps

Operations:

- **insert** - adds a new node to the heap such that the heap property is preserved
- remove maximum element - removes and returns the element at the root and rebuilds the heap property

C++ class for a heap

```
class heap {
private:
    Element* Heap;
    int max_size;
    int curr_size;
    int is_leaf(int);
    int parent(int);
    int left_child(int);
    int right_child(int);
    void sift_down(int );
    void build_heap();
public:
    heap(int);
    heap(Element*, int, int);
    ~heap();
    int heap_size();
    void insert(Element);
    Element remove_max();
};
```

```
int heap::is_leaf(int n){
    return (curr_size/2 <= n) &&
            (n < curr_size);
}

int heap::parent(int n){
    if ((0 < n) && (n < curr_size))
        return (n - 1)/2;
    else
        return OUT_OF_RANGE;
}

int heap::left_child(int n){
    if (2*n+1 < curr_size)
        return 2*n+1;
    else
        return OUT_OF_RANGE;
}
```

```
int heap::right_child(int n){
    if (2*n+2 < curr_size)
        return 2*n+2;
    else
        return OUT_OF_RANGE;
}
```

```
void heap::sift_down(int n){
    while (!is_leaf(n)){
        Element temp; int i, left, right;
        left = left_child(n);
        if ((right = right_child(n)) !=
                                OUT_OF_RANGE)
            if (Heap(left)>Heap(right)
                i = left;
            else
                i = right;
        else
            i = left;
        if (Heap(n) > Heap(i))
            return;
        else{
            temp = Heap(n);
            Heap(n) = Heap(i);
            Heap(i) = temp;
            n = i;
        }
    }}
```

```
void heap::build_heap(){
    for (int i = curr_size/2-1 ; i >= 0 ; i--)
        sift_down(i);
}

heap::heap(int size){
    Heap = new Element(size);
    max_size = size;
    curr_size = 0;
}

heap::heap(Element* array,
                int m_size, int c_size){
    Heap = array;
    max_size = m_size;
    curr_size = c_size;
    build_heap();
}
```

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

int heap::heap_size(){
    return curr_size;
}

void heap::insert(Element v){
    Element temp;
    Heap[curr_size] = v;
    for (int i = curr_size ; (i > 0) &&
            (Heap[i] > Heap[parent(i)]) ;
            i = parent(i)){
        temp = Heap[i];
        Heap[i] = Heap[parent(i)];
        Heap[parent(i)] = temp;
    }
    curr_size++;
    return;
}
```

```
Element heap::remove_max(){
    Element temp;
    if (curr_size > 0){
        curr_size--;
        temp = Heap(0);
        Heap(0) = Heap(curr_size);
        Heap(curr_size) = temp;
        sift_down(0);
        return(Heap(curr_size));
    }
    else
        return OUT_OF_RANGE;
}
```

Priority Queues

- probably the most important application of max heaps
- it stores objects and on request returns the object with the largest value (e.g. jobs in an operating system)

General Trees

Definition. A tree T is a non-empty set of nodes. There is a designated node called the *root* and the other nodes are partitioned into disjoint subsets, which are also trees and the roots of which are children of the root.

Tree traversals

- preorder traversal
- postorder traversal
- inorder does not make sense

General tree ADT

Type: a set of general trees

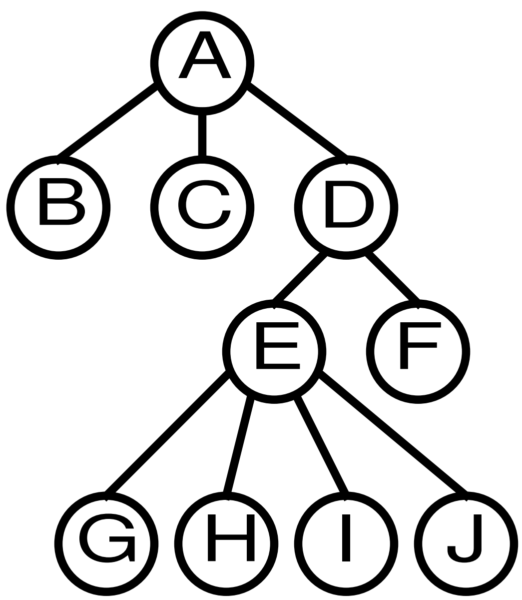
Operations:

- return a node's value
- return a node's parent
- return a node's first child
- return a node's right sibling
- insert a new value as first child
- insert a new value as right sibling
- remove a node

Tree Implementations

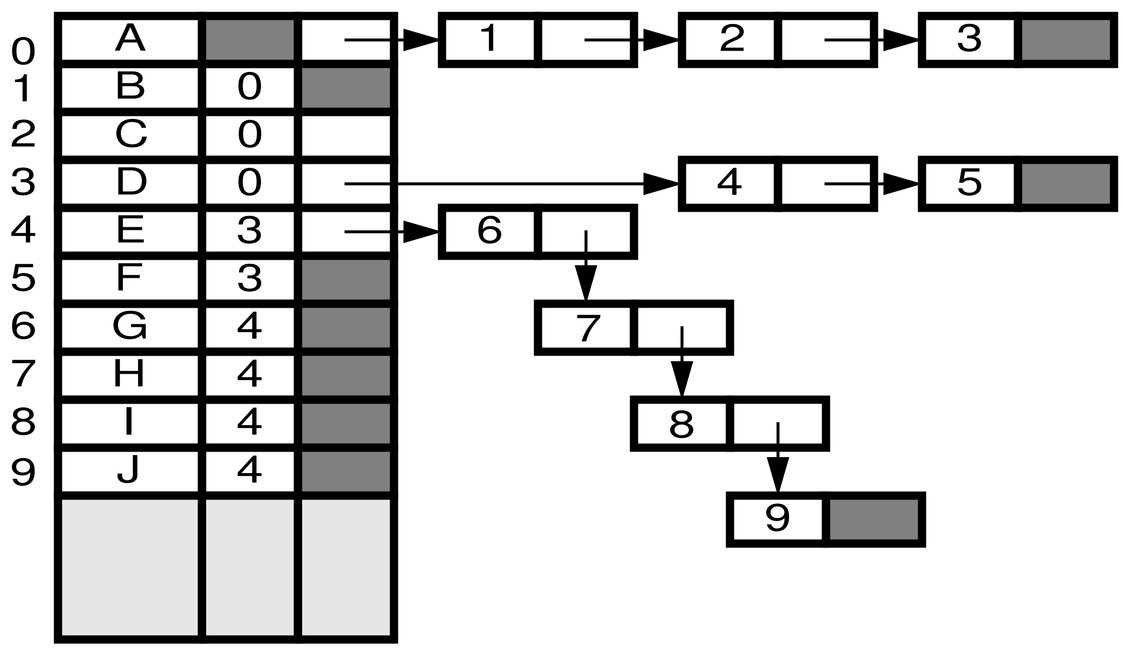
1. Parent pointer implementation

- store for each node the value and a pointer to its parent
- useful for operations such as finding a common ancestor of two nodes



2. List of children

- stores with each node a linked list of children
- it is usually implemented by storing nodes in an array
- each node contains the value, a pointer to its parent and a pointer to the list of its children



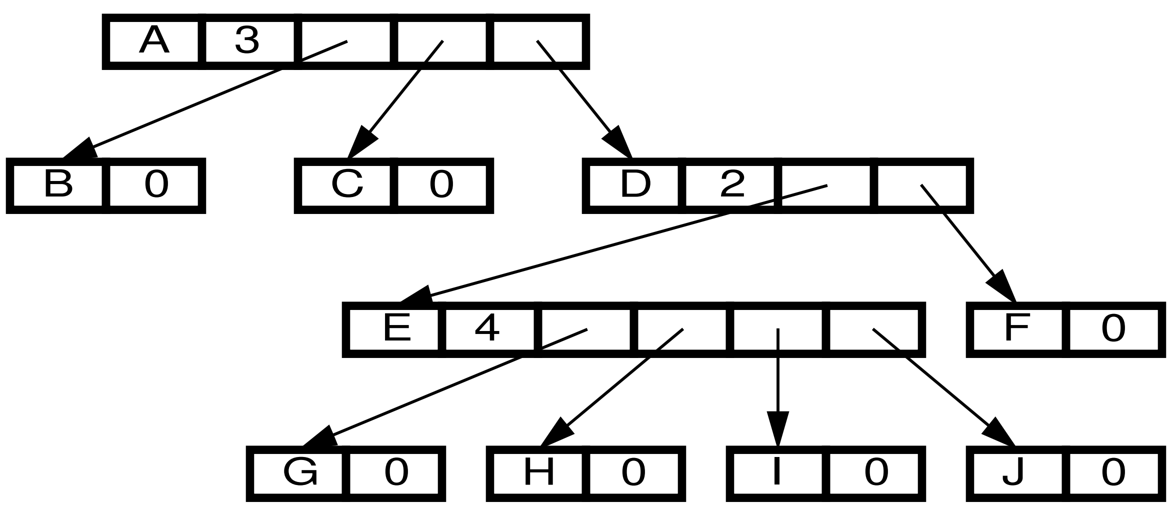
3. Left child/right sibling

- each node stores a value, a pointer to its parent, a pointer to its leftmost child and a pointer to its right sibling

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

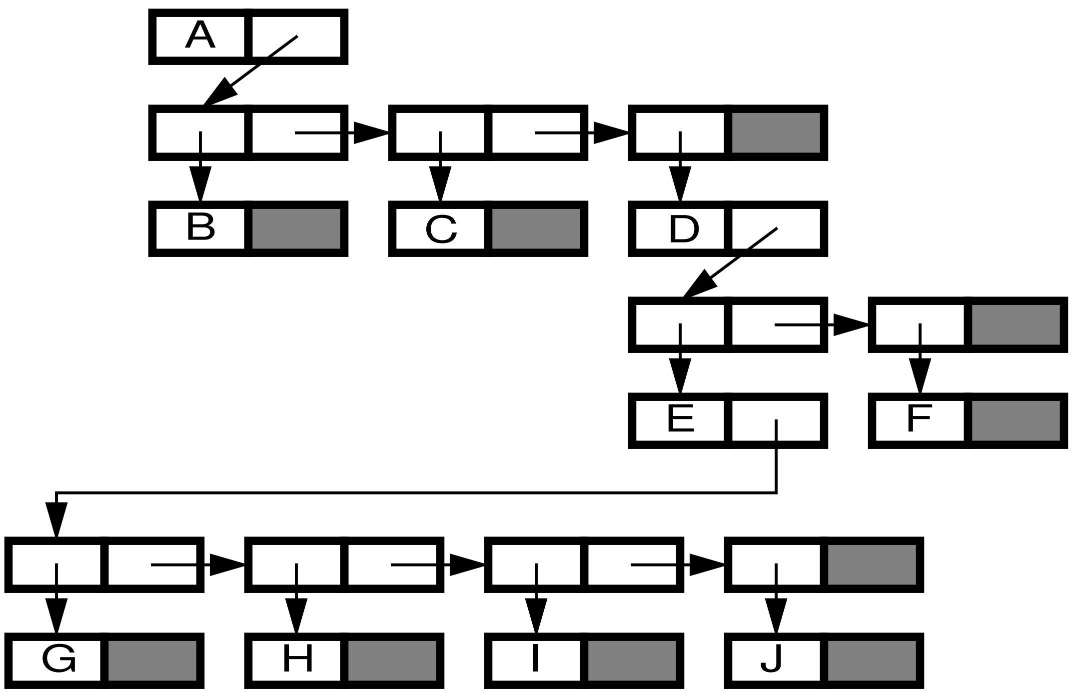
4. Dynamic node

- extends the linked implementation to trees



5. Dynamic left child/right sibling

- substitutes a binary tree for a general tree



6. Sequential implementations

- store the values as they would be implemented by a tree traversal method, plus sufficient information to describe the shap

Example.

- preorder: A(B.C.D(E(G.H.I.J)F))