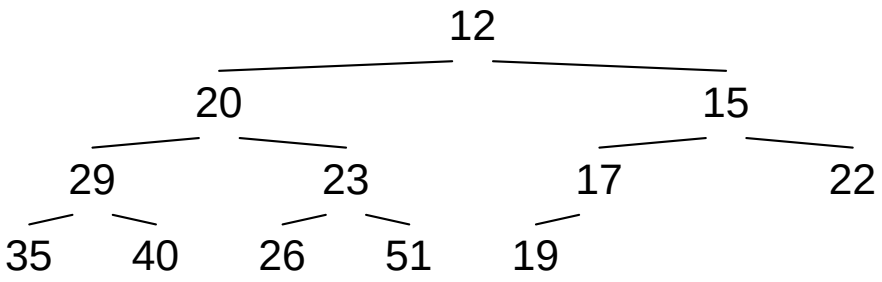
Heaps

The Data Structure
Two Critical Functions
Priority Queues
A Sorting Algorithm

The Data Structure

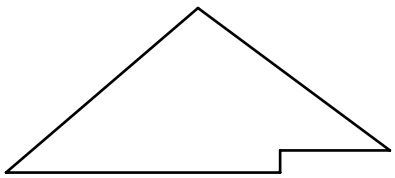
A Heap of Twelve Integers



Two Properties of a Heap

Order: The value at any node is less than or equal to the values of the node's children. (Thus the least element of the set is at the root).

Shape:

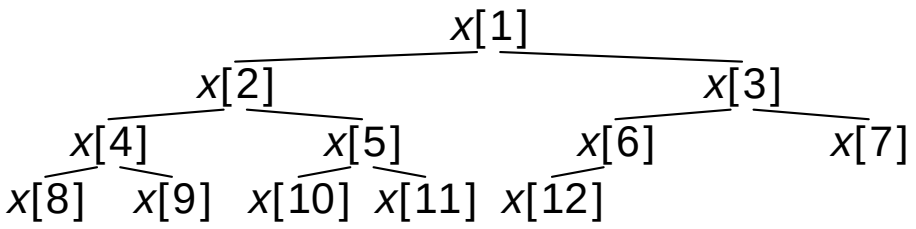


Warning

These heaps all use 1-based arrays

Implementation of Trees with Shape

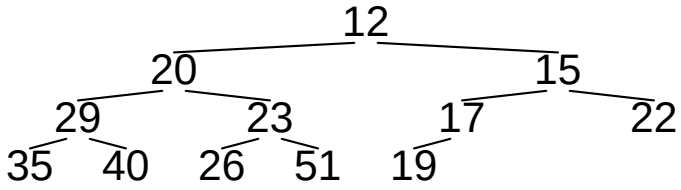
A 12-Element Example

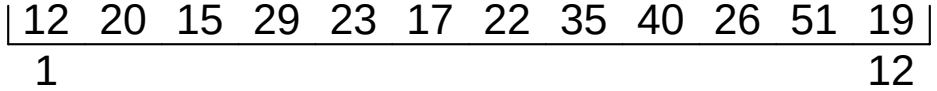


Definitions in a Program

```
root = 1
value(i) = x[i]
leftchild(i) = 2*i
rightchild(i) = 2*i+1
parent(i) = i / 2
null(i) = (i < 1) or (i > n)
```

A Tree and Its Array



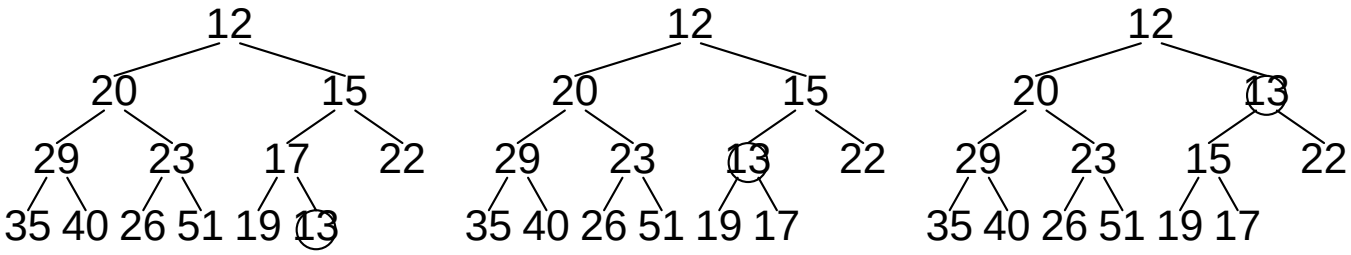


The siftup Function

The Goal

$x[1..n-1]$ is a heap; put a new element in $x[n]$. Sift the new element up the tree.

Inserting 13



The Code

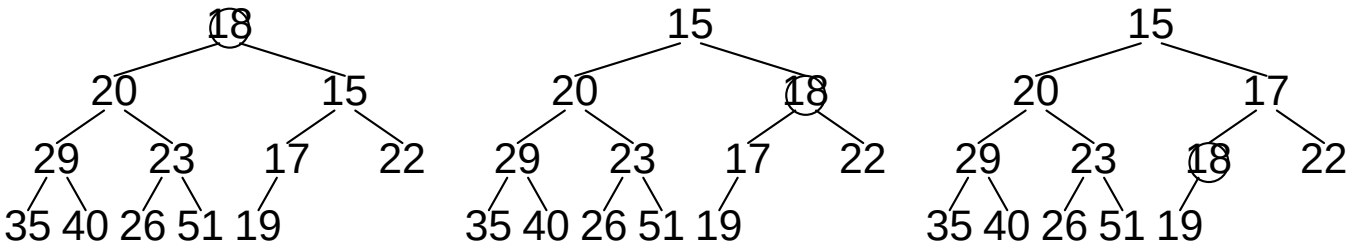
```
void siftup(n)
        pre  n > 0 && heap(1, n-1)
        post heap(1, n)
    i = n
    loop
        /* invariant: heap(1, n) except
            between i and its parent */
        if i == 1
            break
        p = i / 2
        if x[p] <= x[i]
            break
        swap(p, i)
        i = p
```

The siftdown Function

The Goal

x[2..*n*] is a heap; sift *x*[1] down, swapping with the lesser child

Inserting 18



siftdown code

```
void siftdown(n)
        pre   heap(2, n) && n >= 0
        post  heap(1, n)
    i = 1
    loop
        /* inv: heap(1, n) except between
            i and its 0, 1 or 2 kids */
        c = 2*i
        if c > n
            break
        /* c is the left child of i */
        if c+1 <= n
            /* c+1 is the right child of i */
            if x[c+1] < x[c]
                c++
        /* c is the lesser child of i */
        if x[i] <= x[c]
            break
        swap(c, i)
        i = c
```

Priority Queues

The Abstract Data Type

Operations on (initially empty) set *S*.

```
void insert(t)
  pre  |S| < maxsize
  post current S = original S ∪ {t}

int extractmin()
  pre  |S| > 0
  post original S = current S ∪ {result}
       && result = min(original S)
```

Implementations

STRUCTURE	RUN TIMES		
	1 *insert*	1 *extractmin*	*n* of each
Sorted Seq	$O(n)$	$O(1)$	$O(n^2)$
Heaps	$O(\log\ n)$	$O(\log\ n)$	$O(n\ \log\ n)$
Unsorted Seq	$O(1)$	$O(n)$	$O(n^2)$

Heap Implementation of Priority Queues

```
void insert(t)
    if n >= maxsize
        /* report error */
    n++
    x[n] = t
    /* heap(1, n-1) */
    siftup(n)
    /* heap(1, n) */

int extractmin()
    if n < 1
        /* report error */
    t = x[1]
    x[1] = x[n--]
    /* heap(2, n) */
    siftdown(n)
    /* heap(1, n) */
    return t
```

The Complete C++ Class

```
template<class T>
class priqueue {
private:
    int n, maxsize;
    T *x;
    void swap(int i, int j)
    {   T t = x[i]; x[i] = x[j]; x[j] = t; }

public:
    priqueue(int m)
    {   maxsize = m;
        x = new T[maxsize+1];
        n = 0;
    }

    void insert(T t)
    {   int i, p;
        x[++n] = t;
        for (i = n; i > 1 && x[p=i/2] > x[i]; i = p)
            swap(p, i);
    }

    T extractmin()
    {   int i, c;
        T t = x[1];
        x[1] = x[n--];
        for (i = 1; (c = 2*i) <= n; i = c) {
            if (c+1 <= n && x[c+1] < x[c])
                c++;
            if (x[i] <= x[c])
                break;
            swap(c, i);
        }
        return t;
    }
};
```

A Sort Using Heaps

The Idea

Insert into a priority queue, then remove in order

The C++ Code

```
template<class T>
void pqsort(T v[], int n)
{   priqueue<T> pq(n);
    int i;
    for (i = 0; i < n; i++)
        pq.insert(v[i]);
    for (i = 0; i < n; i++)
        v[i] = pq.extractmin();
}
```

Analysis

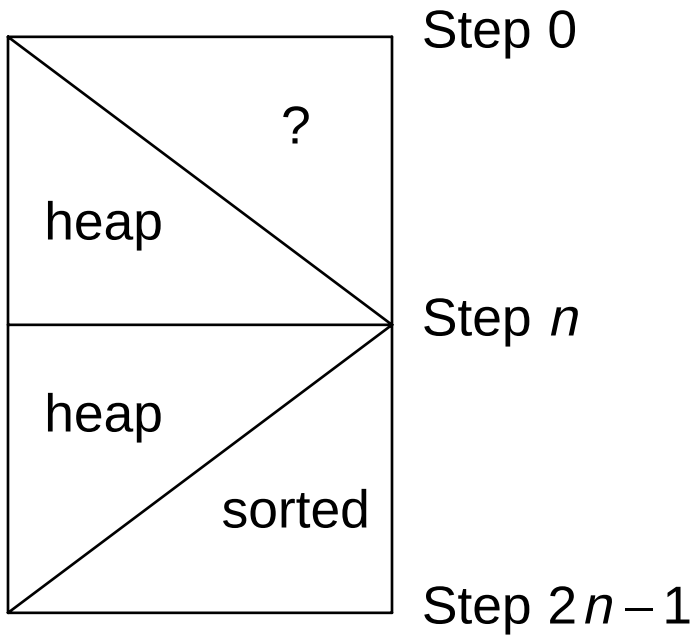
$O(n \log n)$ time

n items of extra space

Heap Sort

The Idea

Use a heap with max on top.



The Code

```
for i = [2, n]
    siftup(i)
for (i = n; i >= 2; i--)
    swap(1, i)
    siftdown(i-1)
```

Heapsort Verification

```
for i = [2, n]
    /* heap(1, i-1) */
    siftup(i)
    /* heap(1, i) */

for (i = n; i >= 2; i--)
    /* heap(1, i)   && sorted(i+1, n)
         && x[1..i]   <= x[i+1..n] */
    swap(1, i)
    /* heap(2, i-1) && sorted(i, n)
         && x[1..i-1] <= x[i..n]   */
    siftdown(i-1)
    /* heap(1, i-1) && sorted(i, n)
         && x[1..i-1] <= x[i..n]   */
```

Principles

Efficiency

shape gives log time for *siftup* and *siftdown*.

Heapsort saves space by overlaying heap and output.

Correctness

Loop invariants.

Invariants of data structures (*shape* and *order*).

Procedural Abstraction

Abstract Data Types