Naive Solution:
- Store items in linear list
- Order?
  - Insert order - fast insert/slow extract
  - Priority order - fast extract/slow insert

Heap: Tree-based structure
(min) heap order: for all nodes, parent’s key ≤ node’s key
[Reverse: max-heap order]

Many variants:
- Binary, leftist, binomial, Fibonacci, pairing, quake, skew... heaps

Binary Heap:
- Simple, elegant, efficient
- Old (1964)
- Basic: insert/extract $O(\log n)$
  - Build: $O(n)$

Priority Queue:
- Stores key-value pairs
  - Key = priority
  - Ops: insert $(x,v)$ - insert value $v$ with key $x$
    - extract-min - remove/return pair with min key value

Priority Queues + Heaps I

Void insert(key x)
    n++; i ← sift-up(n, x)
    A[i] ← x
int sift-up(int i, key x)
    while (i > 1 & & x < A[par(i)])
        A[i] ← A[par(i)]
        i ← par(i)
    return i

Insert(x):
- Append $x$ to end of array
- Sift $x$ up until its parent’s key is smaller (or reaching root)

Heap: Tree-based structure

Pointless tree
Example:

```
Example:

12
18
7
8

3
9

14
24
21
19
```

Binary Heap - Extract Min
- Min key at root → save it
- Copy A[u] to root (A[i]) + decrement n
- Sift the root key down
  - Find smaller of two children
  - If larger, swap with this child
- Return saved root key

Leftist Property:
Null path length

```
npl(v) = \begin{cases} 
-1 & \text{if } v = \text{null} \\
1 + \min \left( npl(v.\text{left}), npl(v.\text{right}) \right) & \text{otherwise}
\end{cases}
```

Def: Leftist Heap is binary tree where:
- Keys are heap ordered
- All nodes v, npl(v.left) ≥ npl(v.right)

Priority Queues & Heaps II

```
Priority Queues & Heaps II
```

Key: extract.min()

```
if (n == 0) Error - Empty heap
result = A[i]
end = A[--n] // get replacement
i = sift-down(i, end)
A[i] = end
return result
```

```
int sift-down(int i, Key z) 
while (left(i) != null)
  u = left(i); v = right(i) 
  if (v <= n && A[v] < A[u])
    u = v // A[u] is smaller child 
  if (A[u] < z)
    A[i] = A[u]; i = u
  else break
return i
```

```
Key: extract.min()
if (n == 0) Error - Empty heap
result = A[i]
end = A[--n] // get replacement
i = sift-down(i, end)
A[i] = end
return result
```

Analysis: Both insert & extract-min take time proportional to tree height
Tree is complete ⇒ O(\log n) time

Leftist Heaps: Meldable heaps
- Can merge two heaps into single heap
- E.g. One processor breaks job. Waiting jobs must be merged with another processor.
Class structure:

```
public class LeftistHeap<Key> {
    private class LHNode { inner class used only by LeftistHeap
        Key x
        LHNode left, right
        int npl
    }
    private LHNode root
    public LeftistHeap() { root = null }
    public void insert(Key x) " Key extractMin() ...
    public function (other private/protected utilities)
    public mergeWith (LeftistHeap H2) { root = merge (this.root, H2.root) H2.root = null helper function merger destroys H2
    ///
    public mergeWith (LeftistHeap H2) { root = merge (this.root, H2.root) H2.root = null helper function merger destroys H2
    
    public functions ⊥ Priority Queues & Heaps III
    
    Merge helper: 2 phases
    1. Merge right paths by order of keys + update npl's
    2. Check leftist property + swap

    Analysis: Time n Rightmost path  = O(\log n) Insert + Extract-min? Exercises
    ```

Lemma: A leftist tree with $r \geq 1$ nodes along its rightmost path has $n \geq 2^{r-1}$ nodes

Proof: (Sketch - see latex notes)

```
T
\rightarrow T
n_u \geq 2^{r-1}, n_v \geq 2^{r-1} \\
\Rightarrow n \geq 2^{r-1} + 2^{r-1} + 2 \Rightarrow n = 2^r - 1 \square
```

Priority Queues + Heaps III

```
LHNode merge(LHNode u, LHNode v) {
    if (u == null) return v
    if (v == null) return u
    if (u.key > v.key) \ // swap so u is smaller
        swap u \leftrightarrow v
    if (u.left == null) u.left \leftrightarrow v
    else
        u.right = merge (u.right, v)
    if (u.left.npl < u.right.npl)
        swap u \leftrightarrow v
    u.npl = u.left.npl + u.right.npl + 1
    return u
```