4. *Heaps*

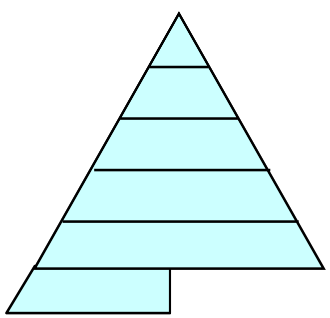
- *d*-heaps
- Leftist heaps
- Fibonacci heaps

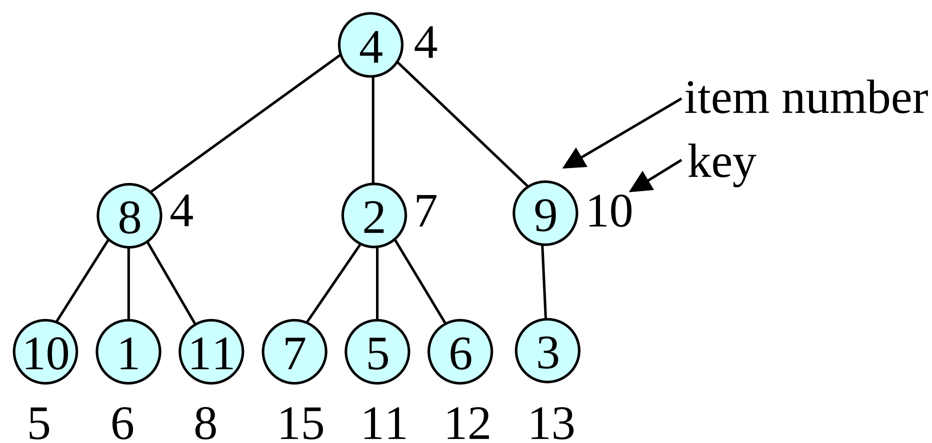
The Heap Data Structure

- To implement Prim's algorithm efficiently, we need a data structure that will store the vertices of *S* in a way that allows the vertex joined by the minimum cost edge to be selected quickly.
- A *heap* is a data structure consisting of a collection of items, each having a key. The basic operations on a heap are:
  - » *insert*(*i*,*k*,*h*). Add item *i* to heap *h* using *k* as the key value.
  - » *deletemin*(*h*). Delete and return an item of minimum key from *h*.
  - » *changekey*(*i*,*k*,*h*). Change the key of item *i* in heap *h* to *k*.
  - » *key*(*i*,*h*). Return the key value for item *i*.
- The heap is among the most widely applicable non-elementary data structure.

d-Heaps

- Heaps can be implemented efficiently, using a *heap-ordered* tree.
  - » each tree node contains one *item* and each item has a real-valued *key*
  - » the key of each node is at least as large the key of its parent (excepting the root)
- For integer d >1, a d-heap is a heap-ordered d-ary tree that is "heap-shaped."
  - » let T be an infinite d-ary tree, with vertices numbered in breadth-first order
  - » a subtree of T is *heap-shaped* if its vertices have consecutive numbers 1, 2, ..., n

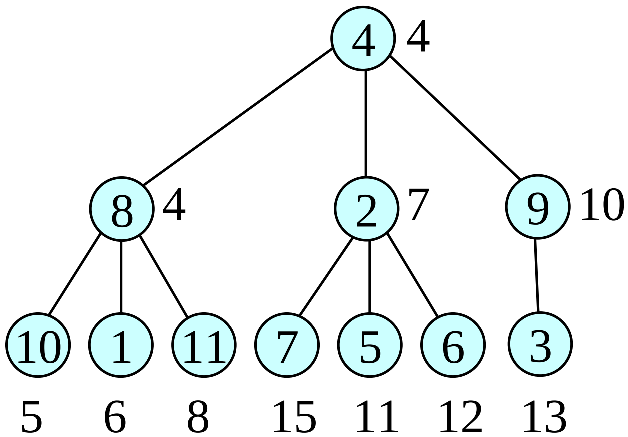




- The depth of a d-heap with n nodes is $\leq \lceil \log_d n \rceil$.

Implementing *d*-Heaps as Arrays

- The nodes of a *d*-heap can be stored in an array in breadth-first order.
  - » allows indices for parents and children to calculated directly, eliminating the need for pointers



	1	2	3	4	5	6	7	8	9	10	11
h:	4	8	2	9	10	1	11	7	5	6	3
key:	6	7	13	4	11	12	15	4	10	5	8

- If i is the index of an item x, then $\lceil (i-1)/d \rceil$ is the index of $p(x)$ and the indices of the children of x are in the range $[d(i-1) + 2 \,..\, di + 1]$.
- When the key of an item is decreased, we can restore heap-order, by repeatedly swapping the item with its parent.
- Similarly, for increasing an item's key.

d-Heap Operations

item function findmin(**heap** h);
   return if $h = \{\} \Rightarrow$ **null**; □ $h \neq \{\} \Rightarrow h(1)$ **fi**;
end;

procedure siftup(**item** i, **integer** x, **modifies heap** h);
   integer p;
   $p := \lceil (x-1)/d \rceil$;
   do $p \neq 0$ **and** $key(h(p)) > key(i) \Rightarrow$
        $h(x) := h(p)$; $x := p$; $p := \lceil (p-1)/d \rceil$;
   od;
   $h(x) := i$;
end;

procedure insert(**item** i; **modifies heap** h);
   siftup(i,$|h| + 1$,h);
end;

```
integer function minchild(integer x, heap h);
   integer i, minc;
   minc := d(x–1) + 2;
   if minc > |h| ⇒ return 0; fi;
   i := minc + 1;
   do i ≤ min {|h|,dx + 1} ⇒
        if key(h(i)) < key(h(minc)) ⇒ minc := i; fi;
        i := i + 1;
   od;
   return minc;
end;
procedure siftdown(item i, integer x, modifies heap h);
   integer c;
   c := minchild(x,h);
   do c ≠ 0 and key(h(c)) < key(i) ⇒
        h(x) := h(c); x := c; c := minchild(x,h);
   od;
   h(x) := i;
end;
```

```
procedure delete(item i, modified heap h);
   item j;  j := h(|h|); h(|h|) := null;
   if  i ≠ j and key(j) ≤ key(i) ⇒ siftup(j,h⁻¹(i),h);
    | i ≠ j and key(j) > key(i) ⇒ siftdown(j,h⁻¹(i),h);
   fi;
end;
item function deletemin(modifies heap h);
   item i;
   if h = {}  ⇒ return null; fi;
   i := h(1); delete(h(1),h);
   return i;
end;
procedure changekey(item i, keytype k, modified heap h);
   item ki;  ki := key(i); key(i) := k;
   if  k < ki ⇒ siftup(i,h⁻¹(i),h);
    | k > ki  ⇒ siftdown(j,h⁻¹(i),h);
   fi;
end;
```

Analysis of *d*-Heap Operations

heap function makeheap(**set of item** s);
  integer j; **heap** h;
  $h := \{\}$;
  for $i \in s \Rightarrow j := |h| + 1;\ h(j) = i;$ **rof**;
  $j = \lceil (|h|-1)/d \rceil$;
  do $j > 0 \Rightarrow$ siftdown($h(j), j, h$); $j = j-1$; **od**;
  return h;
end;

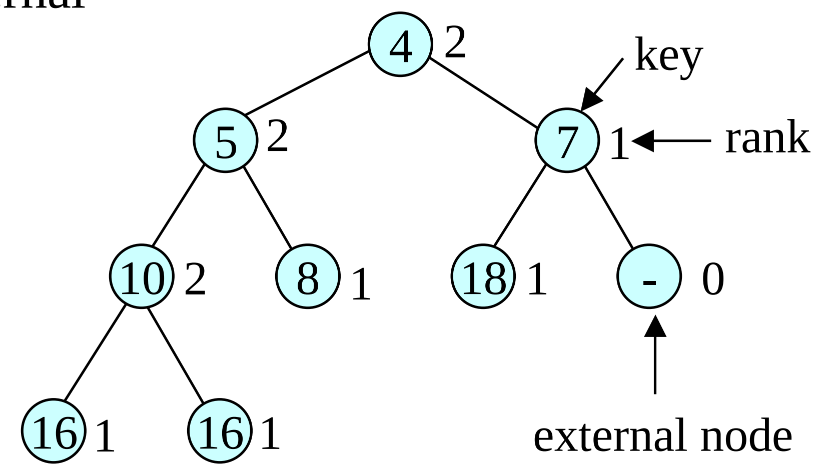
- Each execution of *siftup* (and hence *insert*) takes $O(\log_d n)$ time, while each execution of *siftdown* (and also *delete*, *deletemin*) takes $O(d \log_d n)$ time.
- The time required for *changekey* depends on whether the keys are increased or decreased.
  - if keys are always decreased, we can make *changekey* faster by using a large value for d
- The running time for *makeheap* is $O(f)$ where

$$f(n) = \frac{n}{d}d + \frac{n}{d^2}2d + \frac{n}{d^3}3d + \cdots$$

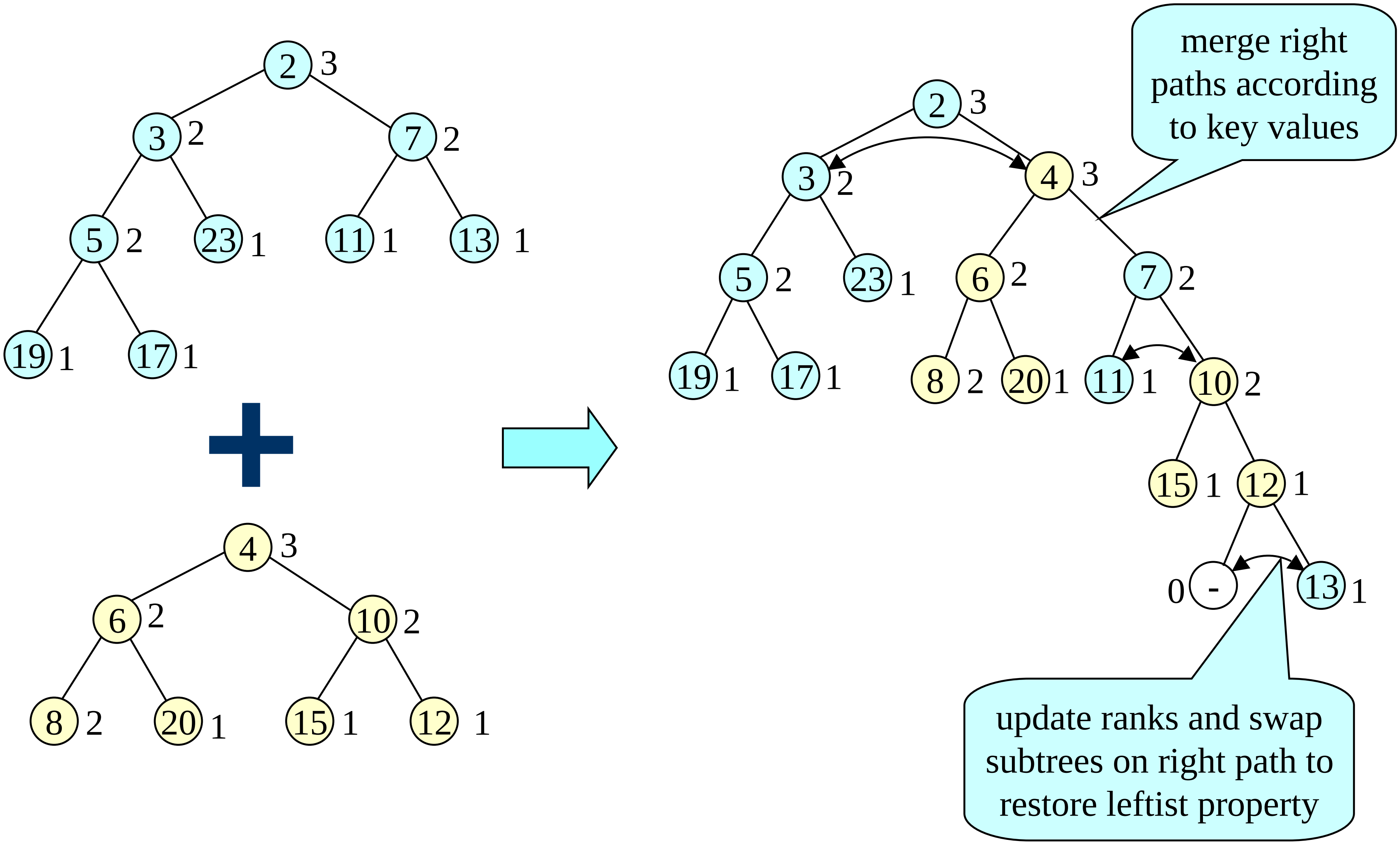
which is $O(n)$.

Leftist Heaps

- The heap operation *meld*(h_1,h_2), which combines the two heaps h_1 and h_2 and returns the resulting heap can't be implemented efficiently using *d*-heaps but can be with an alternative heap implementation known as a *leftist heap*.
  - » if *x* is a node in a *full binary tree*, *rank*(*x*) is defined to be the length of the shortest path from *x* to a leaf that is a descendant of *x*
  - » a full binary tree is *leftist* if *rank*(*left*(*x*)) ≥ *rank*(*right*(*x*)) for every internal node *x*
  - » the *right path* in a leftist tree is path from the root to the rightmost external node
  - » this is a shortest path from root to an external node and has length at most lg *n*
  - » a *leftist heap* is a leftist tree in heap order containing one item per internal node



Melding Leftist Heaps



Implementing Leftist Heaps

heap function meld(**heap** h_1,h_2);

  if h_1 = null $\Rightarrow$ **return** h_2 | h_2 = null $\Rightarrow$ **return** h_1 **fi**;

  if $key(h_1) > key\ (h_2) \Rightarrow h_1 \leftrightarrow\ h_2$; **fi**;

  $right(h_1)$:= meld($right(h_1)$,h_2);

  if $rank(left(h_1)) < rank(right(h_1)) \Rightarrow left(h_1) \leftrightarrow\ right\ (h_1)$ **fi**;

  $rank(h_1) := rank(right(h_1)) + 1$;

  return h_1;

end;

procedure insert(**item** i, **modifies heap** h);

  $left(i) := null$; $right(i) := null$; $rank(i) := 1$;

  h := meld(i,h);

end;

item function deletemin(**modifies heap** h);

  item i; $i := h$;

  h := meld($left(h)$,$right(h)$);

  return i;

end;

Implementing Leftist Heaps in C++

```
typedef int keytyp, oset, item;
class opartition {
        int     n;
        struct node {
                keytyp keyf; int rankf;
                oset leftf, rightf;
        } vec[MAXOSET+1];
public:         opartition(int);
        keytyp  key(item);
        void    setkey(item,keytyp);
        oset    findmin(oset);
        oset    meld(oset,oset);
        oset    insert(item,oset);
        item    deletemin(oset);
        void    print();
        void    sprint(oset);
        void    tprint(oset,int);
};
inline keytyp opartition::key(item i) {
        return vec[i].keyf;
};
```

```
#define left(x)(vec[x].leftf) // etc.
opartition::opartition(int N) {
    n = N;
    for (int i = 1; i <= n ; i++) {
        left(i) = right(i) = Null; rank(i) = 1; key(i) = 0;
    }
    rank(Null) = 0; left(Null) = right(Null) = Null;
}
oset opartition::meld(oset s1, oset s2) {
         if (s1 == Null) return s2;
    else if (s2 == Null) return s1;
    if (key(s1) > key(s2)) { oset t = s1; s1 = s2; s2 = t; }
    right(s1) = meld(right(s1),s2);
    if (rank(left(s1)) < rank(right(s1))) {
        oset t = left(s1); left(s1) = right(s1); right(s1) = t;
    }
    rank(s1) = rank(right(s1)) + 1;
    return s1;
}
```

Heapify

- Operation *heapify*(q) returns heap formed by melding heaps on list q.

  heap function heapify (**list** q);
      if q = [ ] ⇒ **return null fi**;
      do $|q| \geq 2 \Rightarrow q := q[3..]$ & meld($q(1),q(2)$) **od**;
      return $q(1)$
  end

- Let k be the number of heaps on the list initially and let r be the number of heaps on the list after the first $\lceil k/2 \rceil$ melds ($r \leq k/2$).
- Let n_i be the size of the ith heap after the first pass. The time for the first pass is

$$O\left(\sum_{i=1}^{r} 1+\lg n_i\right)$$

- Now, $2 \leq n_i \leq n$ and $\Sigma n_i = n$. Consequently, the time for the first pass is $O(k(1 + \lg(n/k)))$ and the time for the entire *heapify* is

$$O\left(\sum_{j=1}^{\lfloor \lg k \rfloor} (k/2^j)(1+\lg(n2^j/k))\right) = O(k(1+\lg(n/k)))$$

Makeheap and Listmin

- To build a heap in $O(n)$ time from a list of n items,

  heap function makeheap(**set** *s*);
      list *q*; *q* := [ ];
      for $i \in s \Rightarrow$ *left*(*i*),*right*(*i*) := **null**; *rank*(*i*) := 1; *q* := *q* & [*i*]; **rof;**
      return heapify(*q*)
  end;

- Operation *listmin*(*x*,*h*) returns a list containing all items in heap *h* with keys $\leq x$.

  list function listmin(**real** *x*, **heap** *h*);
      if *h* = **null or** *key*(*h*) > $x \Rightarrow$ **return** [ ]; **fi**;
      return [*h*] & listmin(*x*,*left*(*h*)) & listmin(*x*,*right*(*h*));
  end;

  Running time of *listmin* is proportional to number of items listed.

Lazy Melding and Deletion

- It's often possible to improve the performance of algorithms by postponing certain operations.
  - » lazy melding and deletion in leftist heaps postpone melding and deletion
  - » to implement add a *deleted* bit to each node - delete node by setting bit
  - » to meld two heaps, make them children of a dummy node with deleted bit set
  - » remove deleted nodes during *deletemin* operations

item function deletemin(**modifies heap** *h*);
    item *i*;
    h := heapify(purge(*h*)); *i* := *h*; *h* := meld(*left*(*h*),*right*(*h*));
    return *i*
end;

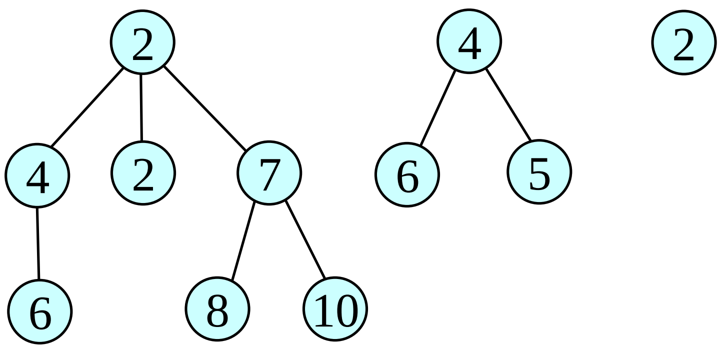
list function purge(**heap** *h*);
    if *h* = **null** ⇒ **return** [ ];
     | *h* ≠ **null and not** deleted(h) ⇒ **return** [*h*]
     | *h* ≠ **null and** *deleted*(*h*) ⇒ **return** purge(*left*(*h*)) & purge(*right*(*h*))
    fi;
end;

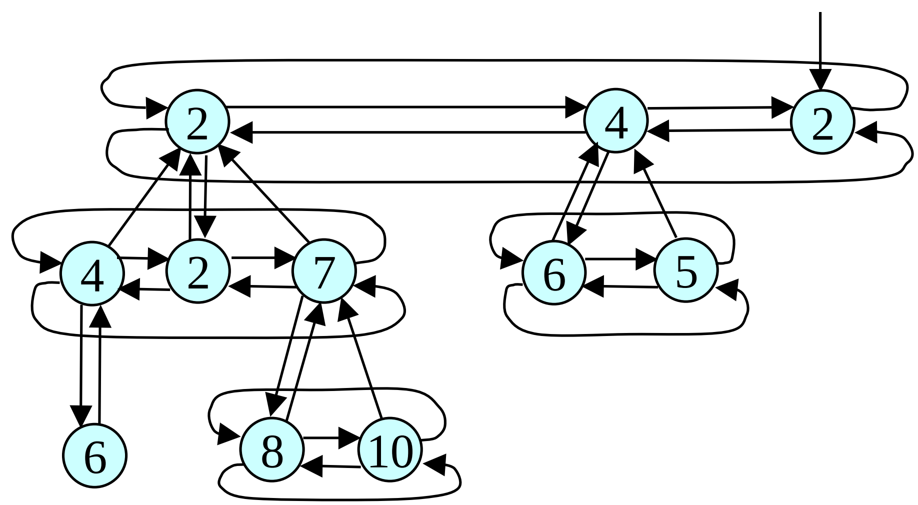
Fibonacci Heaps

- The Fibonacci heap data structure provides an efficient implementation of a collection of heaps.
- *Items* in the heaps are integers over $\{1, \ldots, n\}$. Each item has a key.
- Each non-empty heap is identified by one of its members (its *id*). Initially, there are no heaps.
- Heap operations.
  - » *makeheap*(). Return a new empty heap.
  - » *findmin*(h). Return an item of minimum key in h.
  - » *deletemin*(h). Delete an item of minimum key from h. Return it and the new id.
  - » *meld*(h_1,h_2). Return the id of the heap formed by combining h_1 and h_2. This operation destroys h_1 and h_2.
  - » *decreasekey*(Δ,i,h). Decrease the key of i in h by Δ. Return the new id.
  - » *delete*(i,h). Delete an arbitrary item i from h. Return the new id.

Structure of Fibonacci Heaps

- An F-heap is represented by a collection of heap-ordered trees.
  - » each node has pointers to its parent, its left and right siblings and some child
  - » each node also contains its *key*, an integer *rank* and a *mark* bit
  - » *rank*(*i*) equals the number of children of *i*
  - » the tree roots are linked together on a circular list
  - » the heap is identified by a root node of minimum key





Implementing F-Heap Operations

- To do *meld*, combine root lists.
  - » new heap is identified by the item of minimum key
  - » $O(1)$ time
- To do a *deletemin*, remove the minimum key item from the root list, and combine its list of children with the root list. Then repeat the following step as long as possible
  - » find any two trees with roots of equal rank and link them, making one root the child of the other
- *Deletemin* can be done in *O*(maximum rank + number of linking steps) time.
  - » insert roots into array, at position determined by their rank
  - » combine roots every time there is a collision
  - » note that rank changes when root acquires a new child
  - » initialization trick needed to get indicated running time

Partial Analysis

- We show first that the time to perform a sequence of m operations not including any *delete* or *decreasekey* operations is $O(m+n\log n)$.
- Define the *potential function* for a collection of heaps.
  - » the potential is the number of trees the heaps contain
  - » the potential is zero initially and cannot be negative
- The *amortized time* of an operation is defined to be its actual time plus the net increase it causes in the potential.
  - » the actual time for a sequence of m operations equals the net decrease in potential for the sequence plus the sum of the amortized times of the operations
  - » the time for a sequence of operations is at most the sum of the amortized times
  - » the amortized time for *findmin*, and *meld* is $O(1)$
  - » the actual time for a *deletemin* is $O($ maximum rank+number of linking steps); the amortized time is $O(\log n)$ because each linking step costs one time unit but also decreases the potential by one, and because the ranks are $O(\log n)$ (to be shown).

Decreasekey and Delete

- To perform *decreasekey*(Δ,*i*,*h*)
  - » subtract Δ from *key*(*i*) then cut the edge joining *i* to its parent
  - » make the detached subtree a separate tree in the heap
  - » if *key*(*i*) < *key*(*h*), *i* becomes the minimum node of the heap
  - » increases the potential by 1
- To perform *delete*(*i*,*h*)
  - » perform a *decreasekey* at *i*, that makes *i* the item with smallest key
  - » perform a *deletemin* to remove *i* from the heap
  - » restore the original key value of *i*
  - » time is just sum of the times for the *delete* and *decreasekey* operations

Cascading Cuts

- To keep the ranks from becoming too large, we must add another feature.
  - » let x be a node that becomes a child of some other node because of a linking step
  - » as soon as x loses two children through cuts, the edge to its parent is cut and x becomes the root of a new tree in the heap.
- The mark bits are used to implement this feature
  - » when a node becomes a child of another node through a linking step, its mark bit is cleared
  - » when cutting the edge from a node x to $p(x)$, we decrement the rank of $p(x)$ and check to see if $p(x)$ has a parent
    - if so, set $mark(p(x))$ if it is not set; cut the edge from $p(x)$ to $p(p(x))$ if it is set
    - the cutting procedure is repeated as many times as necessary

Bounds on Ranks

- **Lemma 1**. Let x be any node in an F-heap. Let $y_1, \ldots, y_r$ be the children of x, in order of time in which they were linked to x (earliest to latest). Then, $rank(y_i) \geq i-2$ for all i.
  Proof. Just before y_i was linked to x, x had at least $i-1$ children. So at that time, $rank(y_i)$ and $rank(x)$ were equal and $\geq i-1$. Since y_i is still a child of x, its rank has been decremented at most once since it was linked, implying $rank(y_i) \geq i-2$. ■
- **Corollary 1**. A node of rank k in an F-heap has at least $F_{k+2} \geq \phi^k$ descendants (including itself), where F_k is the kth Fibonacci number, defined by $F_0=0$, $F_1=1$, $F_k=F_{k-1}+F_{k-2}$ and $\phi=(1+5^{1/2})/2$.
  Proof. Let S_k be the minimum possible number of descendants of a node of rank k. Clearly, $S_0=1$, $S_1=2$. By Lemma 1, $S_k \geq 2 + \Sigma_{0\leq i\leq k-2}\, S_i$ for $k \geq 2$. The Fibonacci numbers satisfy $F_{k+2} = 1 + \Sigma_{0\leq i\leq k}\, F_i$ from which $S_k \geq F_{k+2}$ follows by induction on k. ■
- Corollary 1 implies that $rank(x)$ is $O(\log n)$.

Analysis of Fibonacci Heaps

- Define the potential of set of F-heaps to be number of trees plus twice number of marked non-root nodes.
  - » potential is zero initially and cannot be negative
- The amortized time of an operation is defined to be its actual time plus the net increase it causes in the potential.
  - » actual time for a sequence of *m* operations is the net decrease in potential plus the sum of the amortized time of the operations
  - » so, actual time for sequence is less than or equal to amortized time
- Amortized time for *findmin* and *meld* is $O(1)$.
- Actual time for *deletemin* is O(maximum rank+number of linking steps). Amortized time is $O(\log n)$ since a linking step costs one time unit but decreases potential by one and because ranks are $O(\log n)$.
- Actual time for a *decreasekey* is O(number of cuts). Net increase in the potential is at most three minus the number of cascading cuts. So, the amortized time is $O(1)$.
- The amortized time for *delete* is $O(\log n)$.