Chapter 3

Greedy Algorithms

It is the most straightforward technique. It is easy to design and easy to implement. While solving combinatorial optimization problems, we build feasible solutions. In general, a sequence of decisions has to be made.

Greedy algorithms are shortsighted. Every time we make one of these decisions, we look at all the current information and make the least costly choice possible.

This choice is made without any regards to the effect that it might have on later choices. The final feasible solution may or may not be the Optimal. It depends on the problem under investigation.

3.1 Combinatorial Optimization Problems

To introduce (and denote) combinatorial optimization problems, we make use of the following model by Stinson.

Problem instance Some given combinatorial optimization problem

Constraints Some constraints that must be met

Feasible solution A solution of the problem for which the constraints are met

Objective function A function that assigns a non-negative value to each feasible solution

Optimal solution Optimum feasible solution. That is, the feasible solution which maximizes (if profit) or minimizes (if cost) the objective function.

In the next sections, we study problems that can be solved by designing greedy algorithms.

3.2 The 0–1 knapsack problem

We make use of Stinson's framework, discussed in the previous section, to introduce the $0-1$ knapsack problem.

Problem instance

objects	1	2	...	i	...	n
weights	w_1	w_2	...	w_i	...	w_n
profits	p_1	p_2	...	p_i	...	p_n

KNAPSACK with capacity M

Feasible solution

a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ where $x_i \in \{0, 1\}$ with associated weight $W(\mathbf{x}) = \sum_{i=1}^{n} x_i w_i \leq M$.

Objective function

Profit $P(\mathbf{x})$ for $\mathbf{x}$ feasible such that $P(\mathbf{x}) = \sum_{i=1}^{n} x_i p_i$.

Optimal solution

Maximum-profit feasible solution □

The popularity of the 0-1 KNAPSACK is due to:

- it is the simplest integer linear programming problem
- it is often considered as a subproblem to complex ones
- it is a very practical problem
- it is easy to comprehend but difficult to solve

A closely related, but easier to solve problem, is the fractional knapsack problem.

3.2.1 The fractional knapsack problem

In the fractional KNAPSACK problem (or rational KNAPSACK problem), the setup is the same, except that we can take fractions of an object. In other words, a feasible solution is given by a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, where $x_i \in [0, 1]$ (rather than $x_i \in \{0, 1\}$).

3.2.2 The greedy algorithm for the knapsack problem

i.) Rename the objects such that

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_i}{w_i} \geq \cdots \geq \frac{p_n}{w_n}$$

ii.) Place objects $1, 2, \ldots$ updating the cumulative weight at each stage.

iii.) Stop at the j^{th} object if either

a) there is no room for object j:

i) for 0-1 KNAPSACK: $\sum_{i=1}^{j} w_i > M$ and so reject object j

ii) for fractional KNAPSACK: allow fraction of object j to fill up knapsack

or

b) all objects are chosen (i.e. $j = n$, and $w_1 + w_2 + \cdots + w_n \leq M$).

EXAMPLE:

objects	1	2	3	4	5
weight	11	12	8	7	9
profit	23	24	15	13	16

$M = 26$

Note: The objects are in decreasing order (with respect to $\frac{p_i}{w_i}$)
Solving the rational knapsack problem, we get the solution vector

$$\mathbf{x} = (1, 1, \frac{3}{8}, 0, 0)$$

with a total profit of $23 + 24 + \frac{3}{8}(15) = \frac{421}{8}$.

If we use the greedy technique for the 0-1 KNAPSACK problem, we obtain a solution vector

$$\mathbf{x} = (1, 1, 0, 0, 0)$$

with profit $P(\mathbf{x}) = 47$. But $\mathbf{x}' = (1, 0, 1, 1, 0)$ gives a profit of $23 + 15 + 13 = 51$, so the greedy algorithm doesn't give the optimal solution. □

3.2.3 Optimality of the greedy algorithm

Theorem 3.2.1 *The greedy algorithm for the* KNAPSACK *problem always yields the optimal solution when applied to the fractional* KNAPSACK *problem.*

Proof: Suppose that the greedy algorithm generates the solution $\mathbf{x} = (x_1, \ldots, x_n)$, with profit $P = \sum p_i x_i$. Let $\mathbf{y} = (y_1, \ldots, y_n)$ be an optimal solution, with profit $Q = \sum p_i y_i$. Since $\mathbf{y}$ is optimal, we must have $P \leq Q$; we want to prove that $P = Q$.

First observe that if $\sum w_i \leq M$, then the optimal solution is $(1, 1, \ldots, 1)$, and the greedy algorithm generates this solution (no matter what order the objects are considered in). So, we can assume that $\sum w_i > M$. Also, we must have $\sum w_i y_i = M$, or $\mathbf{y}$ cannot be optimal. What we do is show how to transform Q into P, maintaining optimality.

By the way in which the greedy algorithm works, there must be an index k , $1 \leq k \leq n$, such that $x_i = 1$ if $i < k$, $0 \leq x_k < 1$, and $x_i = 0$ if $i > k$. So $\mathbf{x}$ has the form $(1, \ldots, 1, x_k, 0, \ldots, 0)$. Also, observe that we must have $\sum w_i x_i = M$. (This is also a consequence of the way the greedy algorithm works).

Suppose j is the first position in which $x_j \neq y_j$. We distinguish 3 cases:

- Case 1: $j > k$
- Case 2: $j = k$
- Case 3: $j < k$

Case 1: Here $\mathbf{x} = (1, \ldots, 1, x_k, 0, \ldots, 0)$ and $\mathbf{y} = (1, \ldots, x_k, \ldots, y_j, \ldots)$, where $x_k \geq 0$ and $y_j > 0$. This case is in fact impossible, as $\mathbf{y}$ exceeds the knapsack capacity M. We see this as follows:

$$\begin{aligned}\sum_{i=1}^{n} w_i y_i &= \sum_{i=1}^{k} w_i y_i + \sum_{i=k+1}^{n} w_i y_i \\ &= \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i y_i \\ &= M + \sum_{i=k+1}^{n} w_i y_i \\ &> M\end{aligned}$$

since $y_j > 0$.

Case 2: Here $\mathbf{x} = (1, \ldots, 1, x_k, 0, \ldots, 0)$, and $\mathbf{y} = (1, \ldots, 1, y_k, \ldots)$, where $x_k, y_k \geq 0$. First, we must have $y_k < x_k$; for if $y_k > x_k$, then $\mathbf{y}$ exceeds the capacity M by an argument similar to that in Case 1. Hence, $x_k > y_k \geq 0$. Now, since

$$\sum_{i=1}^{n} w_i y_i = \sum_{i=1}^{n} w_i x_i = M$$

and

$$\sum_{i=1}^{k-1} w_i y_i = \sum_{i=1}^{k-1} w_i x_i$$

we have

$$w_k x_k = \sum_{i=k}^{n} w_i y_i$$

or

$$w_k(x_k - y_k) = \sum_{i=k+1}^{n} w_i y_i$$

We also have:

$$\begin{aligned}
Q - P &= \sum_{i=k+1}^{n} p_i y_i - p_k(x_k - y_k) \\
&= \sum_{i=k+1}^{n} \frac{p_i}{w_i} w_i y_i - \frac{p_k}{w_k} w_k(x_k - y_k) \\
&\leq \sum_{i=k+1}^{n} \frac{p_k}{w_k} w_i y_i - \frac{p_k}{w_k} w_k(x_k - y_k) \\
&= \frac{p_k}{w_k} \left(\sum_{i=k+1}^{n} w_i y_i - w_k(x_k - y_k) \right) \\
&= 0
\end{aligned}$$

Hence, $Q \leq P$. Since we also have $Q \geq P$, we must have $Q = P$. Therefore, $\mathbf{x}$ is also an optimal solution.

Case 3: This is the general case; we have $\mathbf{x} = (1, \ldots, 1, x_k, 0, \ldots, 0)$ and $\mathbf{y} = (1, \ldots, 1, y_j, \ldots)$, where $0 \leq y_j < x_j = 1$ (since $j < k$). First, we have

$$\sum_{i=1}^{n} w_i y_i = \sum_{i=1}^{n} w_i x_i$$

or

$$w_j(1 - y_j) + \sum_{i=j+1}^{n} w_i x_i = \sum_{i=j+1}^{n} w_i y_i$$

since $x_j = 1$. Hence,

$$w_j(1 - y_j) \leq \sum_{i=j+1}^{n} w_i y_i$$

This is excess weight available for y_j. We construct a new feasible solution. Choose u_i, $j+1 \leq i \leq n$ such that $0 \leq u_i \leq y_i$ for $j+1 \leq i \leq n$; and so that

$$w_j(1 - y_j) = \sum_{i=j+1}^{n} w_i u_i$$

Now, define $\mathbf{z} = (z_1, \ldots, z_n)$, by

$$\begin{aligned} z_i &= y_i \text{ if } 1 \le i \le j-1 \\ z_i &= 1 \text{ , if } i = j \\ z_i &= y_i - u_i \text{ , if } j+1 \le i \le n \end{aligned}$$

First, we observe that $0 \le z_i \le 1$ for $1 \le i \le n$. Also, we calculate

$$\sum_{i=1}^{n} w_i z_i = \sum_{i=1}^{j-1} w_i y_i + w_j + \sum_{i=j+1}^{n} w_i (y_i - u_i)$$

$$= \sum_{i=1}^{j-1} w_i y_i + w_j - w_j(1 - y_j) + \sum_{i=j+1}^{n} w_i y_i$$

$$= \sum_{i=1}^{n} w_i y_i$$

$$= M$$

so $\mathbf{z}$ is a feasible solution.

Now the profit for $\mathbf{z}$ is

$$R = \sum_{i=1}^{n} p_i z_i$$

We calculate

$$\begin{aligned} R - Q &= \sum_{i=1}^{n} p_i (z_i - y_i) \\ &= p_j(1 - y_j) - \sum_{i=j+1}^{n} p_i u_i \\ &= \frac{p_j}{w_j} w_j (1 - y_j) - \sum_{i=j+1}^{n} \frac{p_i}{w_i} w_i u_i \\ &\ge \frac{p_j}{w_j} w_j (1 - y_j) - \sum_{i=j+1}^{n} \frac{p_j}{w_j} w_i u_i \\ &= \frac{p_j}{w_j} \left(w_j (1 - y_j) - \sum_{i=j+1}^{n} w_i u_i \right) \\ &= 0 \end{aligned}$$

Hence, $R \ge Q$. But $R \le Q$, since $\mathbf{y}$ is optimal. It follows that $R = Q$ and $\mathbf{z}$ is also optimal. Now $z_i = x_i$ for $1 \le i \le j$, so $\mathbf{z}$ agrees with $\mathbf{x}$ in one more coordinate (from the beginning) than $\mathbf{y}$ does. So, we repeat this process as often as necessary, until $\mathbf{y}$ is transformed into $\mathbf{x}$. Since all solutions produced are optimal, $\mathbf{x}$ must also be an optimal solution. This completes the proof.

□

In the next section we study the well-known problem of Huffman codes.

3.3 Huffman codes

Huffman codes are a widely used and very efficient technique for compressing data. They were designed by David Huffman in the early 1950's. Huffman codes are very popular because they

- construct optimal **prefix** codes (a.k.a **prefix–free** or **instantaneous** codes). They are codes in which no codeword is a prefix to a longer codeword
- are very simple
- are often used as part of more elaborate data compression techniques. For example, they are used in data compression standards for fax, speech, image (e.g. JPEG) and video.

Huffman's greedy algorithm uses a table of the frequencies of occurrence of each symbol to build up an optimal way of representing each symbol as a binary string. An optimal code is always represented by a full binary tree (a binary tree, in which every nonleaf node has two children). Hence, given a tree T corresponding to a prefix code, it is always a simple matter to compute the number of bits required to encode a file:
For each character c in the alphabet C, let $f(c)$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree (that is, $d_T(c)$ is the length of the codeword for character c). Thus the number of bits required to encode a file is

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

which we define as the **cost** of tree T.

EXAMPLE:
Suppose we have a 100,000 character file with the following frequencies (the file has only 6 different symbols: $\{a, b, c, d, e, f\}$; see the table).

	a	b	c	d	e	f
frequencies in thousands	45	13	12	16	9	5
fixed-length codeword	000	001	010	011	100	101
variable length codeword	0	100	101	111	1101	1100

Figure 3.1 shows the two trees corresponding to the coding schemes in the above table.

The cost of tree T_1 is 300000 bits ($3 \cdot 100000$), but for tree T_2 only

$$[45 \cdot (1) + 13 \cdot (3) + 12 \cdot (3) + 16 \cdot (3) + 9 \cdot (4) + 5 \cdot (4)]1000 = 224,000$$

bits are required.

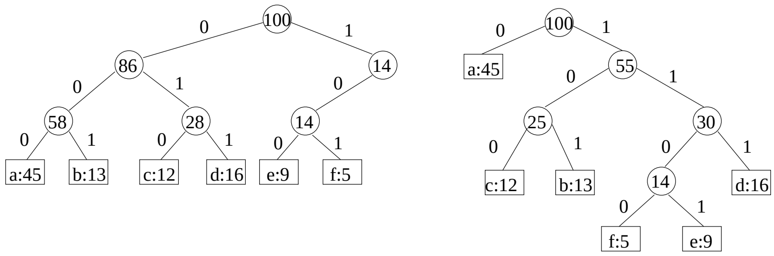


Figure 3.1: The left tree representeds the fixed-length code, the right tree corresponds to the optimal prefix code. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the weights of the leaves in its subtree.

3.3.1 Constructing Huffman Codes

Huffman's greedy algorithm HUFFMAN(C) constructs an optimal prefix code called **Huffman Code** by building the corresponding binary tree T in a bottom-up manner. In the following algorithm, we assume that C is an alphabet of n characters and that for each $c \in C$ the frequency $f[c]$ is given.

```
HUFFMAN(C)
  begin
    n ← |C|
    Q ← C                                  initialize priority queue
    for i ← 1 to n − 1
      do z ← ALLOCATE-NODE()
        x ← left[z] ← EXTRACT-MIN(Q)
        y ← right[z] ← EXTRACT-MIN(Q)
        f[z] ← f[x] + f[y]
        INSERT(Q, z)
    return EXTRACT-MIN(Q)
  end
```

In procedure HUFFMAN(*C*), the priority queue Q is implemented as a binary heap (see below). Therefore, the initialization of Q can be done in $O(n)$ and each (heap-)operation requires $O(\log_2 n)$. Since the loop has $n-1$ iterations, it is performed in $O(n \log_2 n)$, which is the running time of HUFFMAN(*C*).

Remark:

		sum of frequencies
cost of a single merger	=	of the two items
		being merged

So merging the two items with lowest frequencies is a **greedy** step.

We take a short break from the study of greedy algorithms to introduce priority queues that can be used as data structures for the implementation of many algorithms.

3.3.2 Priority queues

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. The operations of a priority queue are:

- INSERT(*S, x*) put element x in S
- MAXIMUM(*S*) returns the element of S with largest key

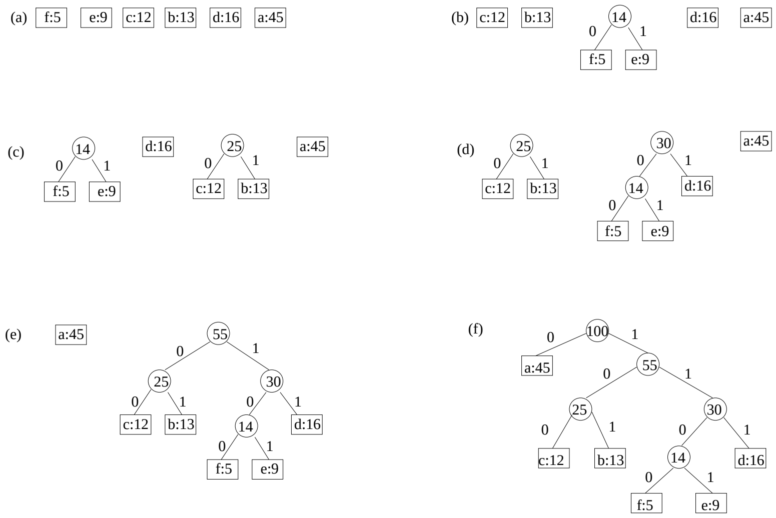


Figure 3.2: The steps of Huffman's algorithm for the frequencies given in the above table. Each step shows the contents of the priority queue sorted into increasing order by frequency. At each step the two trees with lowest frequency are merged.

- EXTRACT-MAX*(S)* removes and returns the element of S with largest key

Note that MAXIMUM and EXTRACT-MAX are often replaced by MINIMUM and EXTRACT-MIN, as in the previous section.

We can implement a priority queue by using **heaps**. The operations on heaps are:

1. HEAP-MAXIMUM, which returns the value $A[1]$ in the heap. Recall that the first entry of the heap A contains the largest element. **Running time:** $\Theta(1)$

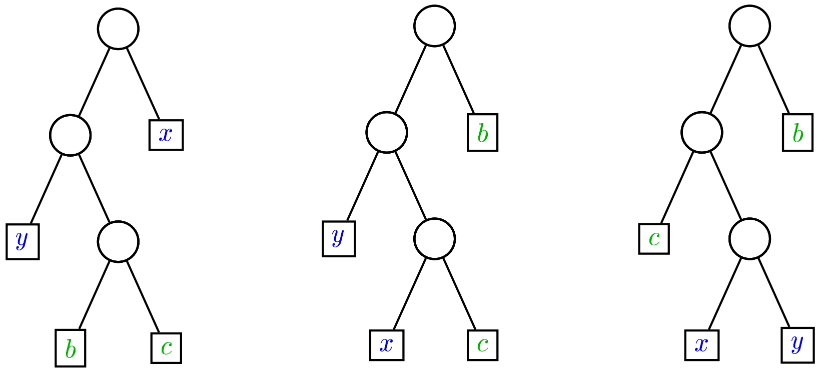
2. HEAP-EXTRACT-MAX, which removes and returns the content of $A[1]$. So it makes use of HEAPIFY.

   HEAP-EXTRACT-MAX*(A)*

   begin
     if heap-size $(A) < 1$
       then "heap underflow"
     $max \leftarrow A[1]$
     $A[1] \leftarrow A[$heap-size $(A)]$
     heap-size $(A) \leftarrow$ heap-size $(A) - 1$
     HEAPIFY*(A,1)*
     return max
   end

   Running time: $O(\log_2 n)$ which, recall, is the amount of work needed to heapify.

3. HEAP-INSERT, which inserts a node into heap A by first adding a new leaf then traversing the tree (only one path), bottom up, to find the proper place for the new element.



T
Optimal Tree

T'
Interchange x and b

T''
Interchange y and c

Figure 3.3: The trees T, T' and T''

HEAP-INSERT*(A, key)*

begin
  heap-size(A) $\leftarrow$ heap-size(A)+1
  $i \leftarrow$ heap-size (A)
  while $i > 1$ **and** $A[parent(i)] < key$
    do $A[i] \leftarrow A[parent(i)]$
    $i \leftarrow parent(i)$
  $A[i] \leftarrow key$
end

Running time: $O(\log_2 n)$ which is the cost of going up the tree.

We present two ingredients that are generally present in problems that can be generally solved by greedy algorithms.

3.3.3 Elements of the greedy strategy

Two ingredients that are present in most problems that lend themselves to a greedy strategy:

Greedy choice property
: a globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

Optimal substructure
: an optimal solution to the problem contains within it optimal solutions to subproblems.

We now use this principle to prove the correctness of Huffman's algorithm.

3.3.4 Correctness of Huffman's algorithm

To show that Huffman's procedure yields a prefix code with minimum cost, we have to show:

1. **Greedy-choice property**
   Show that merging the lowest two frequency characters does not cause the tree to become non-optimal. We show that any optimal tree can be reorganized such that the two characters (with lowest frequency) have the same parent.

② **Optimal substructure**
After making an optimal first choice, the problem is reduced to finding a Huffman code for a smaller alphabet.

Proof:

① **Greedy-choice property**
Let x and y be two characters in C having lowest frequencies. We can always construct an optimal prefix code for C in which the encodings for x and y have the same length and differ only in the last bit. We assume that $f(x) \leq f(y)$. Suppose that T is an optimal tree with $f(b) \leq f(c)$ where b and c are siblings of maximum depth. We thus have: $f(x) \leq f(b)$ and $f(y) \leq f(c)$. We construct a tree T' by interchanging x and b and then we exchange the positions in T' of y and c to produce a tree T'' (see Figure 3.3). We'll show that T and T'' have equal costs:

$$\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f(x) d_T(x) + f(b) d_T(b) - f(x) d_{T'}(x) - f(b) d_{T'}(b) \\
&= f(x) d_T(x) + f(b) d_T(b) - f(x) d_T(b) - f(b) d_T(x) \\
&= \underbrace{(f(b) - f(x))}_{\substack{\geq 0 \\ x \text{ has lowest} \\ \text{frequency} \\ \text{(as } x \text{ and } y \\ \text{are of lowest} \\ \text{frequency)}}} \cdot \underbrace{(d_T(b) - d_T(x))}_{\substack{\geq 0 \\ b \text{ is a leaf} \\ \text{of maximum} \\ \text{depth in T}}} \geq 0
\end{aligned}$$

Thus $B(T) - B(T') \geq 0$. But $B(T) \leq B(T')$ since $B(T)$ is the cost of the optimal tree T.
Therefore $B(T) = B(T')$.

Similarly, we can show that

$$B(T') = B(T'')$$

and thus

$$B(T) = B(T'')$$

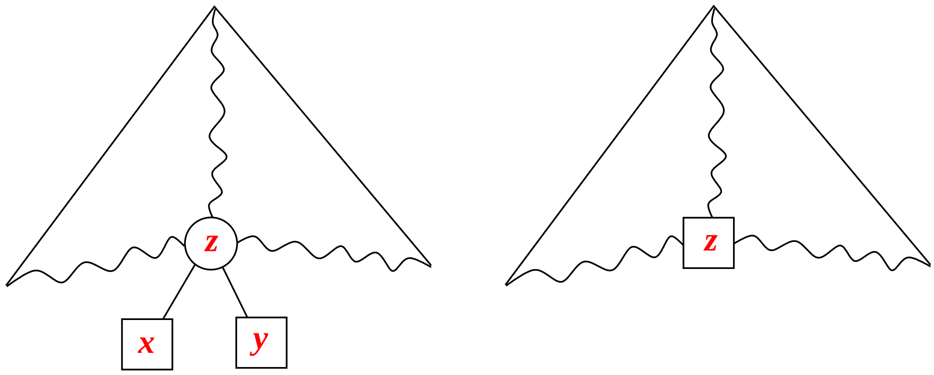
□

② **Optimal substructure property**
Show that T', the tree obtained from T by removing the two leaves x and y (see Figure 3.4), represents an optimal prefix code for alphabet $C' = C - \{x, y\} \cup \{z\}$. For each $c \in C - \{x, y\}: d_T(c) = d_{T'}(c)$. Consequently

$$f(c) d_T(c) = f(c) d_T(c) \qquad \forall\ c \in C - \{x, y\}$$

Note: $d_T(x) = d_T(y) = d_{T'}(z) + 1$.
So

$$\begin{aligned}
f(x) d_T(x) + f(y) d_T(x) &= (f(x) + f(y))(d_{T'}(z) + 1) \\
&= f(z) d_{T'}(z) + (f(x) + f(y)) \\
\Rightarrow B(T) &= B(T') + f(x) + f(y)
\end{aligned}$$



T: full binary tree representing optimal prefix code over C

$T' = T \setminus \{x, y\}$ where z has frequency $f(z) = f(x) + f(y)$

Figure 3.4: Optimal substructure property

Suppose T' is not optimal for C'. Then there exists T'' whose leaves are characters in C' such that:

$$B(T'') < B(T')$$

Since z is treated as a character in C', it appears as a leaf in T'. Now consider the tree T''' that is obtained by attaching x and y to T'' by making them children of z. Note that T''' represents a prefix code for C.

$$\begin{aligned} B(T''') &= B(T'') + f(x) + f(y) \\ &< B(T') + F(x) + f(y) \text{ since } B(T'') < B(T') \\ &= B(T) \end{aligned}$$

So we obtained $B(T''') < B(T)$, contradicting the optimality of T.
Therefore T' is optimal for C'.

3.4 The general greedy procedure

Most problems amenable to greedy techniques have algorithms that are very similar to GREEDY*(C)*.

```
GREEDY(C)                                       C is a set of candidates
  begin
    S ← ∅                                       we construct the
                                                solution in the set S
    while (C ≠ ∅) and ¬SOLUTION(S)
      x ← SELECT(C)
      C ← C−{x}
      if FEASIBLE((S) ∪ {x})
        then S ← S ∪ {x}
    if SOLUTION(S)
      then return S
      else return "no solution"
  end
```

Remark: The SELECTION() procedure is related to the objective function.
Next we consider scheduling problems.

3.5 Minimum tardy task problem

We start by giving an EXAMPLE:

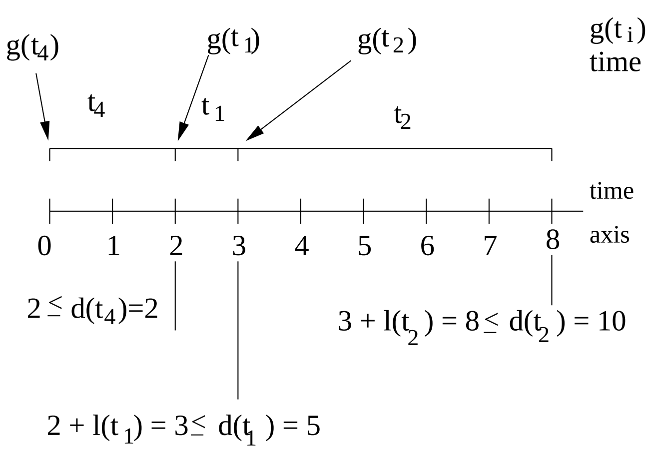
$$T = \{t_1, t_2, t_3, t_4\}$$

is a set of tasks. We denote by:

- $l(t)$: length it takes to accomplish task t
- $w(t)$: weight of task t (penalty)
- $d(t)$: deadline by which task t should be done

	t_1	t_2	t_3	t_4
$l(t_i)$	1	5	3	2
$d(t_i)$	5	10	5	2
$w(t_i)$	3	2	5	4

Consider $T_1 = \{t_1, t_2, t_3\}$. We want to schedule the tasks of T_1 on a "uniprocessor". Since $d(t_4) < d(t_1) < d(t_2)$ it makes sense to schedule t_4 first, followed by t_1 and then t_2.



So $T_1 = \{t_1, t_2, t_4\}$ is a feasible solution. Let W denote the sum of the weights of the unscheduled tasks (i.e. the tardy task weight).

$$W = \sum_{t \in (T - T_1)} w(t)$$

In our example $T - T_1 = \{t_3\}$ and therefore $W = w(t_3) = 5$. □

Note that at most one task is executed at a time, and each scheduled task finishes before its deadline.

The minimum tardy task problem is a minimization problem that has a maximization counterpart. Just replace the weights by profits and now the objective function is the sum of the profits of the scheduled tasks.

Remark:

$S \subseteq T$ is feasible $\Leftrightarrow$ the tasks in S can be scheduled in increasing order by deadline without violating any deadline

In other words, if $|T| = n$ and $S \subseteq T$ with $|S| = k$, we only have to check **one** of the $k!$ possible schedules for the feasibility of S.

3.6 Scheduling with Deadlines

Once again we use Stinson's framework to introduce the scheduling with deadlines problem.

Problem instance

tasks	1	2	...	i	...	n in T	
lengths	l_1	l_2	...	l_i	...	l_n	$l_i > 0$
deadlines	d_1	d_2	...	d_i	...	d_n	$d_i > 0$
profits	p_1	p_2	...	p_i	...	p_n	$p_i > 0$

Feasible solution

any schedule of $S \subseteq T$ such that:

$$g : S \to Z^+ \cup \{0\}$$

where

i) $g(i) < g(j) \Rightarrow g(i) + l_i \leq g(j)$

ii) $g(i) + l_i \leq d_i$

Objective function

a profit $P = \sum\limits_{i \in S} p_i$

Optimal solution

maximum-profit schedule S

Example:

Problem instance

tasks	1	2	3	4	5	6	7	8
length	2	4	1	7	4	3	5	2
deadlines	3	5	6	8	10	15	16	20
profits	15	20	16	19	10	25	17	18

- $S = \{1, 3, 5, 6\}$
  S is feasible with $P = 66$
- $S' = \{2, 3, 4, 6, 8\} \Rightarrow g(4) + l_4 = 5 + 7$ and $d_4 = 8$ i.e. $g(4) + l_4 > d_4$
  Therefore S' is infeasible. □

Remark: If $l_1 = l_2 = l_3 = \ldots = l_n = 1$ then the problem can be solved in polynomial time. An obvious greedy algorithm for constructing S is to add at each step the job with the highest value of p_i among those not yet considered, provided that the chosen set of jobs remains feasible.

```
Min-Tardy-Task(T)
  begin
    rename tasks such that p_1 ≥ p_2 ≥ ... ≥ p_n
    S ← ∅
    for i = 1 to n
      if S ∪ {t_i} is feasible
      then S ← S ∪ {t_i}
    return P = Σ_{t∈S} p(t)
  end
```

Note that the tasks in $S \cup \{t_i\}$ are first sorted by increasing deadline and then scheduled . Obviously this can be a very inefficient algorithm unless we use appropriate data structures in the implementation.

Remark: MIN-TARDY-TASK*(T)* gives the optimal solution when $l_1 = l_2 = \ldots = l_n = 1$.
The next topic we tackle is the problem of finding minimum spanning trees (MST).

3.7 Minimum Spanning Trees (MST)

An acyclic graph is a graph that has no cycles. If *any* 2 distinct nodes have an edge between them then the graph is complete. A tree is an acyclic, connected graph. A spanning tree of a graph G is a tree that is a subgraph of G and contains every vertex of G. A spanning tree of minimal total weight, i.e. the sum of the weights of all edges is minimum, is called a minimum spanning tree (MST). We study two algorithms for finding minimum spanning trees: Kruskal's and Prim's.

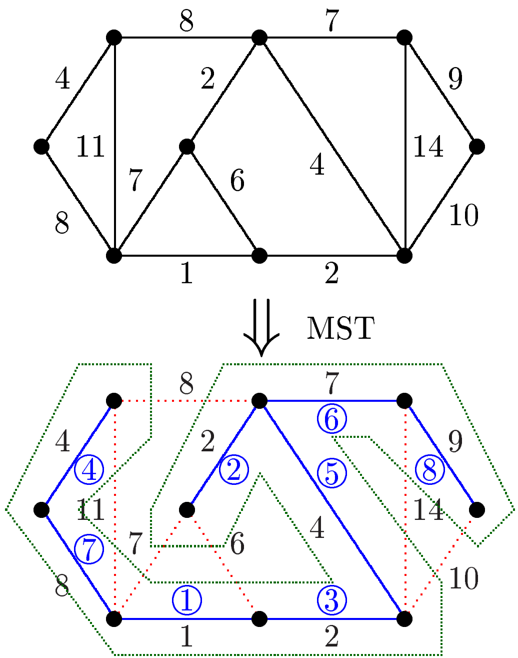
3.7.1 Kruskal's MST Algorithm

Given a weighted graph, at each step Kruskal's algorithm chooses the lowest-weighted remaining edge form anywhere in the graph, discarding any edge that would form a cycle with those already chosen. At any time the edges chosen so far will form a forest but not necessarily one tree.

Kruskal's procedure uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices of a tree. The operations needed are:

- FIND-SET*(u)*: returns a representative element from the set that contains u.
- UNION*(u, v)*: combines the two trees that contain u and v.

EXAMPLE: Given the following graph, Kruskal's method will construct the MST depicted in the second graph. The tree is in bold and the order in which the edges are chosen are indicated by circled numbers.



```
KRUSKAL
begin
    A ← ∅
    for each vertex v ∈ V                    creates |V| trees each one
        do MAKE-SET(v)                       containing one vertex
    sort edges of E by
    nondecreasing weight w
    for each edge (u,v) ∈ E, in order
        by nondecreasing weight
        do if FIND-SET(u) ≠ FIND-SET(v)      u and v are from different trees
              then A ← A ∪ {(u,v)}
                   UNION(u, v)
    return A
end
```

The running time of Kruskal's algorithm depends on the implementation of the disjoint-set data structure:

- initialization takes $O(|V|)$
- sorting the edges takes $O(|E| \log_2 |E|)$
- there are $|E|$ iterations of the loop (i.e. on disjoint-set data structure). If the disjoint-set forest data structure (see next subsection) with the union by rank and path compression procedures is used, then the work performed for the loop is $O(|E|\alpha(|E|, |V|))$ where α is the functional inverse of ackermann's function (see [CLR90, Section 22.4]).

  Since $\alpha(|E|, |V|) = O(\log_2 |E|)$, the total running time of Kruskal's algorithm is $O(|E| \log_2 |E|)$.

In the next section we give a brief overview of the disjoint-set data structure that can be used to implement Kruskal's algorithm.

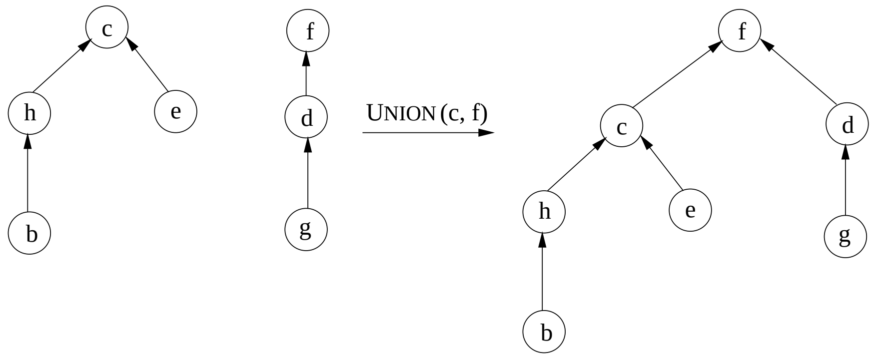
3.7.2 Disjoint-set data structure

A disjoint-set data structure maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets. Each set is identified by a representative, which is just some member of the set. We maintain the sets as rooted trees. The operations required are:

- MAKE-SET*(x)*: Creates a new set, whose only member (and thus representative) is pointed to by x. Note that x is not in any other set since they are disjoint sets.
- UNION*(x, y)*: Unites the sets that contain x and y to form a new set. Note that the 2 sets were disjoint prior to the call to UNION*(x, y)*.
- FIND-SET*(x)*: Returns a pointer to the representative of the (unique) set containing x. If the sets are represented by trees, then FIND-SET*(x)* consists in going up the tree to the root node. Visited nodes are called "find path".

Disjoint-set forests

Each member points to its parent. The root contains the representative and is its own parent.



To efficiently implement the operations of disjoint-set forests, we make use of two heuristics:

1. **Union by rank**: Make the tree with fewer nodes point to the root of the tree with more nodes. With each node, we associate a rank. $Rank[x]$ is an upper bound on the number of edges in the longest path between x and a descendant leaf (i.e., on the depth of the subtree rooted at x).

2. **Path compression**: Make each node visited during FIND-SET (i.e. nodes that belong to the “find path”) point directly to the root. This is done to speed up future FIND-SET operations. Note that path compression does not alter the ranks of the nodes.

We now introduce the procedures associated with the disjoint-set data structure.

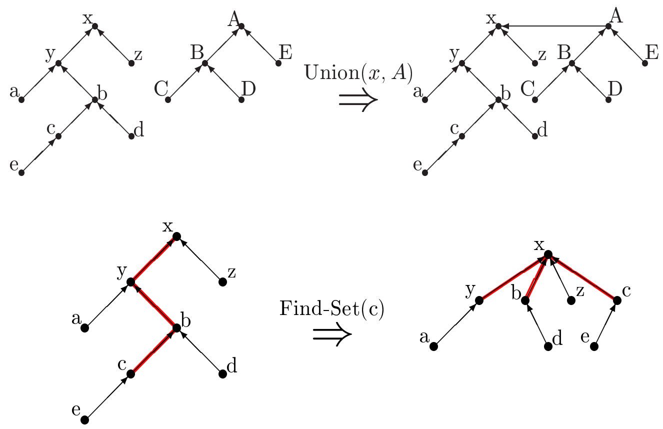
```
MAKE-SET(x)
  begin
    parent[x] ← x
    rank[x] ← 0
  end
```

```
UNION(x, y)
  begin
    LINK(FIND-SET(x),FIND-SET(y))
  end
```

```
LINK(x, y)
  begin
    if rank[x] > rank[y]
      then parent[y] ← x
      else parent[x] ← y
          if rank[x] = rank[y]
            then rank[y] ← rank[y] + 1
  end
```

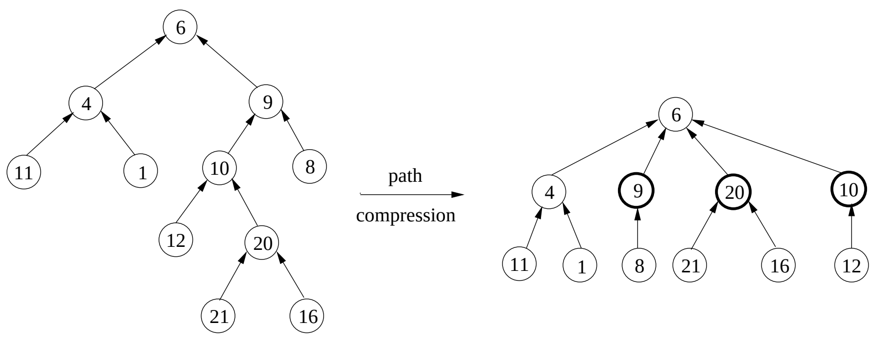
```
FIND-SET(x)                                   makes 2 passes
                                              for path compression
  begin
    if x ≠ parent[x]
      then parent[x] ← FIND-SET(parent[x])
    return parent[x]
  end
```

EXAMPLE for UNION and FIND-SET



□

EXAMPLE for path-compression



□

Figure 3.5: FIND-SET(20): nodes in the find-path (i.e. $9, 10, 20$) now point directly to the root. The remaining nodes are not touched.

We now study the second MST algorithm mentioned in the introduction of this subsection.

3.7.3 Prim's minimum spanning tree algorithm

Prim's algorithm selects an arbitrary starting vertex, then branches out from the tree constructed so far by choosing a new vertex and edge at each iteration. The choice is done **greedily**. During the course of the algorithm, the vertices are divided into three disjoint sets:

- red vertices: nodes in the tree constructed so far
- blue vertices: nodes not in the tree but adjacent to some red node
- black vertices: nodes that are neither red nor blue

Convention: An edge between a red node and a blue node is a violet edge. An edge in the tree is colored red.

PrimColor MST algorithm

1. color an arbitrary node red, the rest blue and black

2. **while not** (all nodes red)
   do
   select the shortest violet edge;
   color it and its blue endpoint red;
   update colors of nodes and edges
   end

Remark: The current red nodes and red edges form a subtree of a minimum spanning tree.

With efficient implementation that involves "good" data structures the above algorithm can have as running time

$$O(|E| + |V| \log_2 |V|)$$

(which is a big improvement over the $O(|V|^3)$ behavior exhibited by the above algorithm). We now give a more formal study of Prim's algorithm.

Prim's MST algorithm

Given a weighted graph, at each step Prim's algorithm chooses the lowest-weighted edge among the ones that have an end-point in the tree constructed so far.

Recall: The algorithm selects an arbitrary starting vertex, then branches out from the tree constructed so far by choosing a new vertex and edge at each iteration.

Procedure PRIM takes a connected graph G and the root r (starting vertex) as input. All vertices not in the tree constructed so far are in a priority queue Q based on a key field. $\mathsf{key}[v]$ is the minimum weight of any edge connecting v to a vertex in the tree. $\mathsf{key}[v] = \infty$ if there is no such edge. $\pi[v]$ names the "parent" of v in the tree. The MST will be:

$$\{(v, \pi[v]) | v \in V - \{r\}\} \text{ after executing } \textsc{Prim}(G, r)$$

```
PRIM(G, r)
  begin
    Q ← V
    for each u ∈ Q
      do key[v] ← ∞
    key[r] ← 0
    π(r) ← NIL                          since root r has no parent
    while Q ≠ ∅ do
        u ← EXTRACT-MIN(Q)
        for each v ∈ Adj[u] do
          if (v ∈ Q ∧ w(u,v) < key[v])
            then π[v] ← u
                 key[v] ← w(u,v)
  end
```

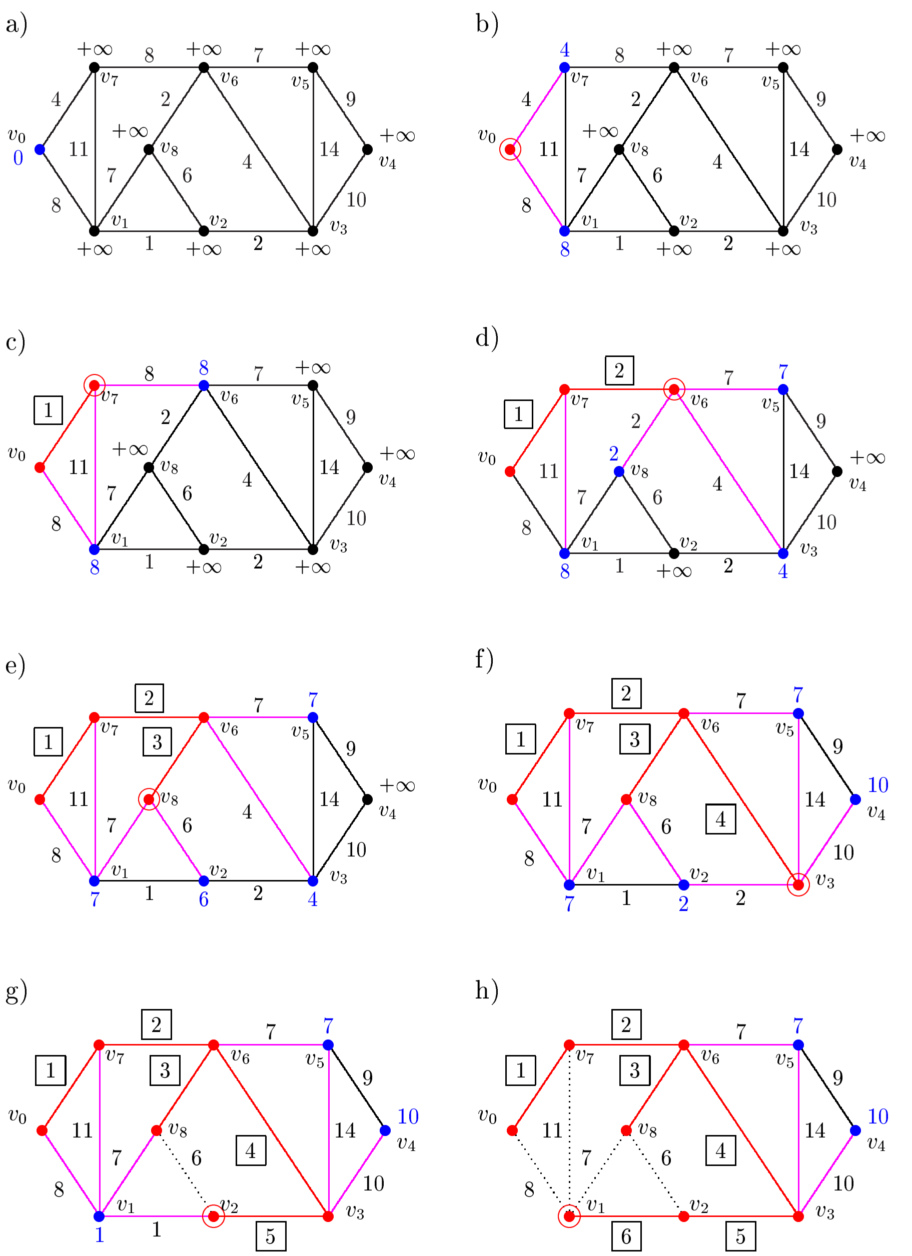
Vertices are removed one by one from Q to form the tree. If Q is implemented as a binary heap, BUILD-HEAP can be used for the initialization in $O(|V|)$ time. The total work for the loop is $|V|$ and since each EXTRACT-MIN operation takes $O(\log_2 |V|)$, $O(|V| \log_2 |V|)$ calls for EXTRACT-MIN are required. The inner **for** loop is executed $O(|E|)$ times, and $\mathsf{key}[v] \leftarrow w(u,v)$ is an $O(\log_2 |V|)$ operation on the heap (the test for membership in Q can be implemented in constant time by keeping a bit in every vertex that tells whether it is in Q or not, and updating the bit when the vertex is removed from Q). Therefore we obtain a total running time of

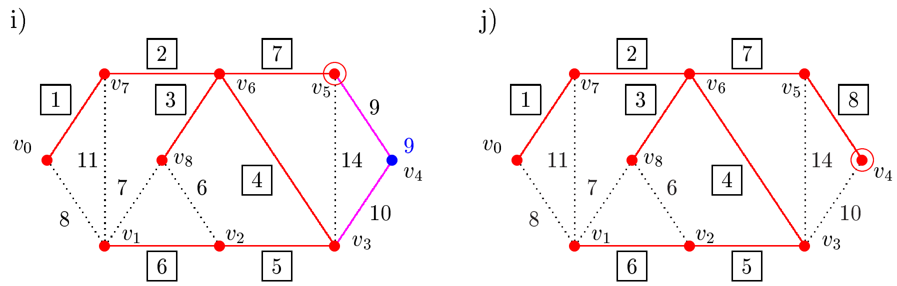
$$O(|V|\log_2 |V| + |E|\log_2 |V|) = O(|E|\log_2 |V|)$$

for Prim's algorithm. By using Fibonacci Heaps as priority queues, the running time can be improved to

$$O(|E| + |V|\log_2 |V|)$$

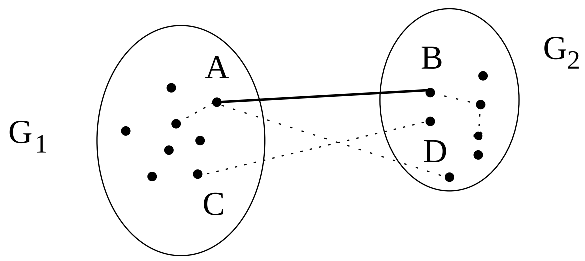
Example: The following figures (a)–(j) show the execution of Prim's algorithm.





The greedy algorithms by Kruskal and Prim work due to following.

Theorem 3.7.1 *Given any division of the vertices of a graph into two set G_1 and G_2, the MST of the graph contains the shortest of the edges with one endpoint in G_1 and the other in G_2.*



Proof (*by contradiction*): Suppose that $< A, B >$ is the shortest edge with one endpoint A in G_1 and the other endpoint B in G_2.
Suppose that T is the MST of the graph and $< A, B > \notin T$. Consider $T \cup \{< A, B >\}$.

$T \cup \{< A, B >\}$ contains a cycle since it now has n edges. Note that T contains an edge $< C, D >$ with $C \in G_1$ and $D \in G_2$. Consider $T' = T \cup \{< A, B >\} - \{< C, D >\}$. T' is a spanning tree of the graph and $w(T') < w(T)$ since $w(< A, B >) < w(< C, D >)$. But this contradicts the minimality of T. Therefore $< A, B >$ must be in T. □

Once again, we pause from greedy algorithms for a while to consider more sophisticated priority queues.

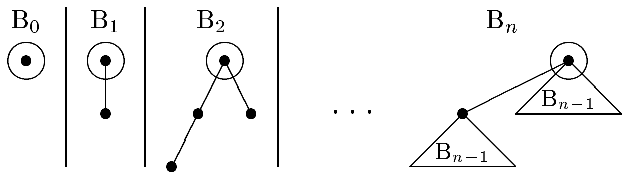
3.7.4 Binomial Heaps and Fibonacci Heaps

Ordinary binary heaps (as used in heapsort) are adequate to be used as priority queues as long as we don't need to merge trees. Merging (via UNION operation) uses heapify and therefore takes $\Theta(n)$ time in the worst case.

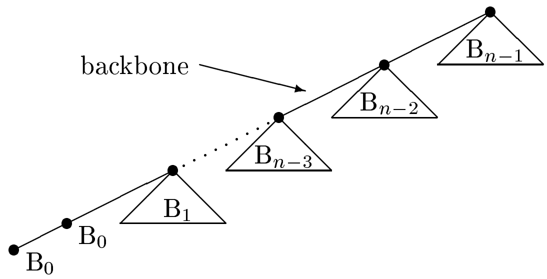
We now explore 2 priority queues that have better running times when performing the operation of merging trees. These two data structures – Binomial Heaps and Fibonacci Heaps (see Section 3.7.6) – are also known as mergeable heaps.

3.7.4.1 Binomial Heaps

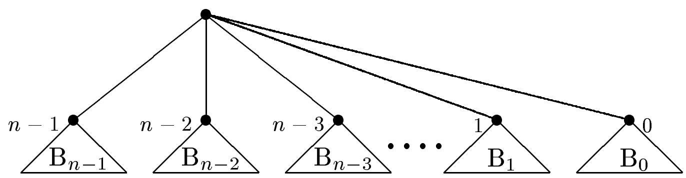
A Binomial Heap is a collection of Binomial trees. Binomial trees are defined recursively by:



Another way of looking on binomial trees may be:



or



Properties of B_n

1. *B_n has 2^n nodes.*
   Proof (*by induction*): B_0 has $2^0 = 1$ node. Assume that B_{k-1} has 2^{k-1} nodes. B_k consists of 2 copies of B_{k-1}, each having 2^{k-1} nodes (our induction hypothesis). Therefore B_k has $2^{k-1} + 2^{k-1} = 2^k$ nodes.

2. *The height of B_n is n.* Use induction on n.

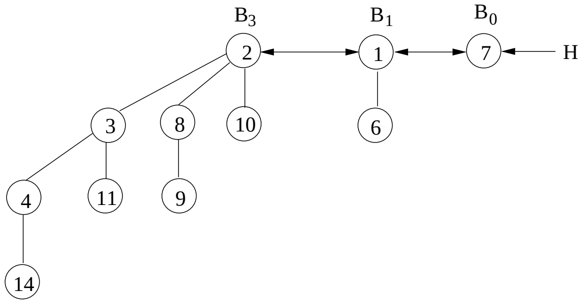
3. *The number of nodes of B_n at level k is the binomial coefficient $\binom{n}{k}$ (hence the name binomial tree).*
   Observation: Let $D(n, k)$ be the number of nodes of B_n at level k. A node at level k in B_{n-1} appears in B_n once at level $k+1$ (left subtree) and once at level k (right subtree).
   Proof (*by induction on n*):
   B_0 has 1 node at level 0 ($\binom{0}{0} = 1$). Suppose the statement is true for B_{t-1}. For B_t, we have $D(t, i) = D(t-1, i) + D(t-1, i-1)$ (by above observation). But $D(t-1, i) = \binom{t-1}{i}$ and $D(t-1, i-1) = \binom{t-1}{i-1}$ by induction hypothesis. So $D(t, i) = \binom{t-1}{i} + \binom{t-1}{i-1} = \binom{t}{i}$. Therefore, there are $\binom{n}{k}$ nodes at level k of B_n.

4. *The root node of B_n has degree n which is greater than that of any other node. If the children of the root node are numbered from left to right by $n-1, n-2, \ldots, 0$, then child i ($0 \le i \le n-1$) is the root node of subtree B_i.* It can be proven by using induction on n.

Remark: The maximum degree of any node in an n-node binomial tree is $\log_2 n$.

A **binomial heap** H is a set of binomial trees that satisfies the following binomial-heap properties:

EXAMPLE:



□

Figure 3.6: The binomial heap H has 11 nodes. $11 = (1011)_2$. So B_3, B_1 and B_0 are present with $8 + 2 + 1 = 11$ nodes.

1. Each binomial tree in H is heap-ordered: the key of a node is greater than or equal to the key of its parent. So the root node contains the smallest key in the tree.

2. There is at most one binomial tree in H whose root has a given degree. A binomial heap containing n nodes has at most $\lfloor \log_2 n \rfloor + 1$ binomial trees. The binary representation of n has $\lfloor \log_2 n \rfloor + 1$ bits b_i where

$$n = \sum_{i=0}^{\lfloor \log_2 n \rfloor} b_i 2^i$$

Binomial tree B_i appears in the binomial heap H	$\Leftrightarrow$	Bit $b_i = 1$

Recall: B_i has 2^i nodes.
So the binomial heap H contains at most $\lfloor \log_2 n \rfloor + 1$ binomial trees.

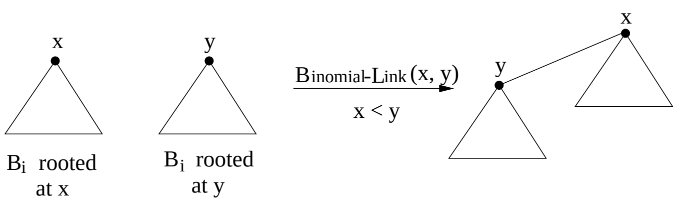
An example of a binomial heap is shown in Figure 3.6.

Operations on binomial heaps
Recall that a binomial heap is a collection of binomial trees of different sizes, each having the heap property (heap-ordered). So the minimum key is stored in one of the root nodes. The operations are:

① Creating a new binomial heap is an $\Theta(1)$ operation. The procedure returns an empty heap H.

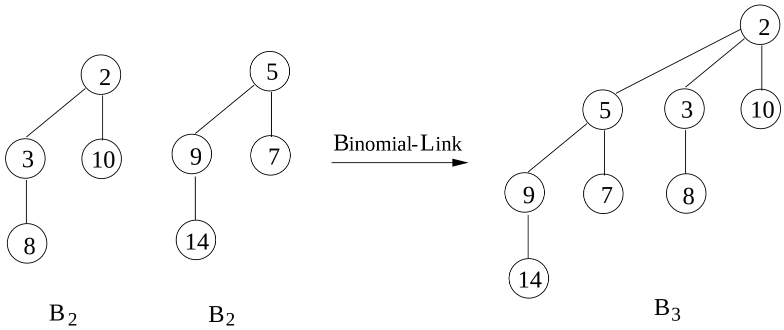
② Finding the minimum key is an $O(\log_2 n)$ operation, since we have to check the $\lfloor \log_2 n \rfloor + 1$ roots in the worst case.

③ Uniting two binomial heaps: two binomial heaps of different sizes can be united by using the procedure BINOMIAL-LINK that combines two binomial trees of equal sizes.



Make the root with the larger value (here y) into the i^{th} child of the root of the smaller value to form B_{i+1}. Growth rate $O(1)$.

EXAMPLE:



□

BINOMIAL-HEAP-UNION*(H_1, H_2)* merges two heaps H_1 and H_2, each consisting of a collection of binomial trees arranged in increasing order of size. Merging is done very much like binary addition. At each stage we have up to three ones, one carried from the previous position, and one from each operand:

- if there are none $\Rightarrow$ result: 0
- if there is just one $\Rightarrow$ result: 1
- if there are two ones $\Rightarrow$ result: 0+ carry 1 to next position
- if there are three ones $\Rightarrow$ result: 1+ carry 1 to next position

BINOMIAL-HEAP-UNION*(H_1, H_2)* first looks for B_0 in H_1 and H_2.

- if there are no B_0's $\Rightarrow$ do nothing at this stage
- if there is one B_0 $\Rightarrow$ keep it to form part of the result
- if there are two B_0's $\Rightarrow$ use BINOMIAL-LINK to merge them into B_1.

Now look for B_1's. There can be a maximum of three B_1's.

- if there is none $\Rightarrow$ do nothing
- if there is one B_1 $\Rightarrow$ keep it to form part of the result
- if there are two B_1's $\Rightarrow$ merge them into B_2
- if there are there B_1' s $\Rightarrow$ merge two of them into B_2; keep the third B_1 to form part of the result

and so on with B_2, B_3, ... As we go on, we join the roots of the binomial trees that are kept to form part of the result. We also keep track of the root with the smallest value (i.e. keep updating MIN). The merging operation has at most $\lfloor \log_2 n \rfloor$ stages each requiring at most one BINOMIAL-LINK which can be done in $O(1)$. So the complete merge takes $O(\log_2 n)$.

4. Inserting a node is done by BINOMIAL-HEAP-INSERT(H, x). Item x will be inserted in the binomial heap H by first creating a one element heap H' containing only x, and then using BINOMIAL-HEAP-UNION(H_1, H_2) to merge it with H. Creating H' takes $\Theta(1)$ as seen previously and merging H and H' takes $O(\log_2 n)$ again as already shown. So inserting a node takes $O(\log_2 n)$.

5. BINOMIAL-HEAP-EXTRACT-MIN*(H)* extracts the node with the minimum key. BINOMIAL-HEAP-EXTRACT-MIN performs the following steps:

EXAMPLE: Consider the following two binomial heaps H_1 and H_2.

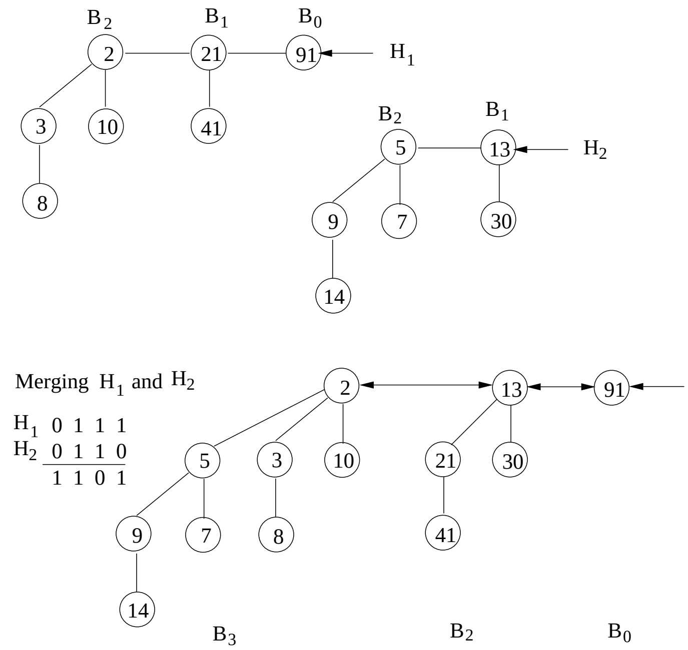


Figure 3.7: EXAMPLE for BINOMIAL-HEAP-UNION

□

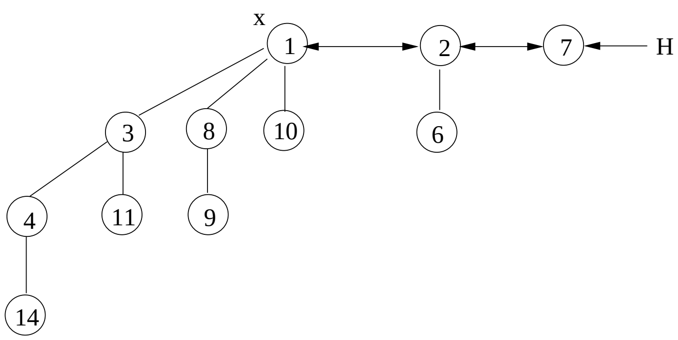
a) find the root x with minimum value and remove its binomial tree B from H, joining any parts of H into a new binomial heap H

b) remove x form B. Join the binomial trees that used to be the subtrees of x into a new binomial heap H_2.

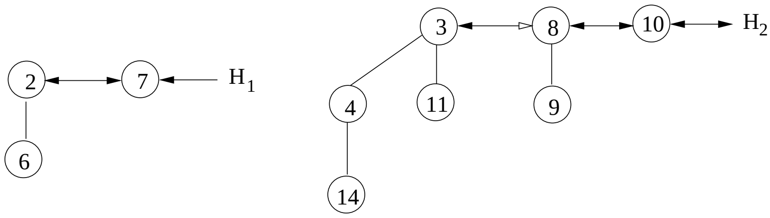
c) merge H_1 and H_2

Once again, extracting the node with minimum key is an $O(\log_2 n)$ operation.

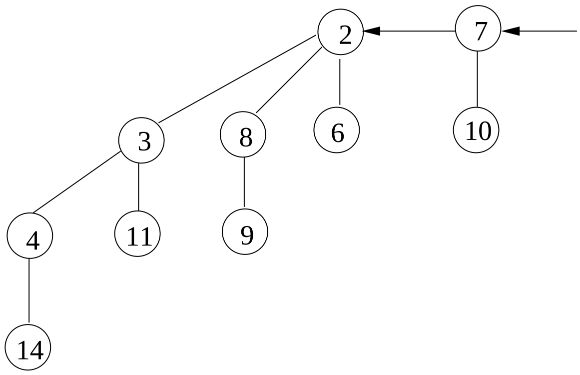
EXAMPLE for BINOMIAL-HEAP-EXTRACT-MIN*(H)*



after removing x from B_3 we get



after merging H_1 and H_2



□

⑥ Decreasing a key: BINOMIAL-HEAP-DECREASE-KEY*(H, x, k)* decreases the value of the node x in binomial heap H to a new value k. The procedure finds x, gives it its new value k and now checks to see if the heap-order is preserved. To do that, it just has to go up the tree as is the case with binary heaps. Going up the tree is an $O(\log_2 n)$ operation, since the maximum depth of x is $\lfloor \log_2 n \rfloor$

⑦ Deleting a key is performed by BINOMIAL-HEAP-DELETE*(H, x)*. BINOMIAL-HEAP-DELETE calls BINOMIAL-HEAP-DECREASE-KEY*(H, x, $-\infty$)* forcing x to "bubble up" to the root of its tree.
Then a call to BINOMIAL-HEAP-EXTRACT-MIN*(H)* will now remove x. Again the growth rate here is $O(\log_2 n)$.

We now study a new analyzing technique that is needed for a better understanding of algorithms that have parts that perform repetitive steps.

3.7.5 Amortized Analysis

In order to get a better understanding of Fibonacci Heaps, we need a new analyzing technique: **amortized analysis**. The amortized cost is the theoretical speed of a given set of operations.

It is an elegant method for analyzing the running times of certain algorithms in a more precise way. If a certain part of the algorithm is performed several times, each time with a different running time, then, instead of taking the worst case every time, we amortize the different costs; we get a tighter bound than the one produced by the worst case analysis. The bound produced by the worst case analysis is sometimes too pessimistic.

In an amortized analysis, the time required to perform a sequence of data structure operations is averaged over all the operations performed. It is used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation might be expensive.

The amortized cost is $O(f(n))$ when the worst case execution time of an entire sequence of n operations never exceeds $O(nf(n))$.

Amortized analysis is different from average-case analysis. There is no probability involved. We are averaging over a sequence of operations, not the possible running times of a single operation. The averaging process is called *amortization.* It is a clever way used by accountants to average large one-time costs over long periods of time. There are several different methods for deriving amortized bounds for a sequence of operations.

A. The aggregate method

We show that, for all n, a sequence of n operations takes worst-case time $T(n)$ in total. So in the worst case, the average cost, or the amortized cost, per operation is $\frac{T(n)}{n}$.
EXAMPLE: Incrementing a binary counter
$A[0 \dots k]$ is an array of k bits representing a number X where

$$X = \sum_{j=0}^{k-1} A[j]2^j$$

So $A[0]$ holds the low-order bit, $A[k-1]$ the high-order bit. Assume $length(A) = k$. Consider procedure INCREMENT*(A)* that adds **one** to the binary representation of X.

```
INCREMENT(A)
  begin
        j := 0
        while ((j < length(A)) ∧ (A[j] = 1))
              A[j] := 0
              j := j + 1
        if j < length(A)
              then A[j] := 1
  end
```

Note that $A[j] = 1 \Leftrightarrow 2^j$ appears in the sum $X = \sum_{j=0}^{k-1} A[j]2^j$. The running time of procedure INCREMENT depends on the value of X, i.e. the contents of array A. If the first t bits of A are all ones, then INCREMENT takes $\Theta(t)$ time to flip the $t+1$ bits. So in general, if X is a number between 0 and n, INCREMENT takes $\Theta(\log_2 n)$ time in the worst-case, since the binary representation of n is $\lfloor \log_2 n \rfloor$ bits long.

Suppose we want to use INCREMENT to count from 0 to n. Start with $X = 0$, in other words

$$A[0] = A[1] = \dots A[n-1] = 0$$

The worst-case running-time analysis yields $O(n \log_2 n)$ since we assume $\Theta(\log_2 n)$ per iteration. But do we flip $\Theta(\log_2 n)$ bits every time we call INCREMENT ?

j	bits that are flipped	total number of flips
1	$A[0]$	1
2	$A[0], A[1]$	2
3	$A[0]$	1
4	$A[0], A[1], A[2]$	3
5	$A[0]$	1
6	$A[0], A[1]$	2
7	$A[0]$	1
8	$A[0], A[1], A[2], A[3]$	4
9	$A[0]$	1
10	$A[0], A[1]$	2
11	$A[0]$	1
12	$A[0], A[1], A[2]$	3
13	$A[0]$	1
14	$A[0], A[1]$	2
15	$A[0]$	1
16	$A[0], A[1], A[2], A[3], A[4]$	5
total number of flips for $X = 0$ to $X = 16$ is 31		

We easily see that:

- $A[0]$ flips every time (or $\lfloor \frac{16}{1} \rfloor$ times)
- $A[1]$ flips every other time (or $\lfloor \frac{16}{2} \rfloor$ times)
- $A[2]$ flips every 4^{th} time (or $\lfloor \frac{16}{4} \rfloor$ times)
- $A[3]$ flips every 8^{th} time (or $\lfloor \frac{16}{8} \rfloor$ times)
- $A[4]$ flips every 16^{th} time; the next time it will flip will be for $X = 32$

In general: $A[j]$ flips every $(2^j)^{th}$ time (or $\lfloor \frac{16}{2^j} \rfloor$ times). So if we start with an array of zeros ($X = 0$), a sequence of n increments flips each bit $A[j]$ exactly $\lfloor \frac{n}{2^j} \rfloor$ times. Hence the total number of bits flipped for the entire sequence (counting from 0 to n) is :

$$\sum_{j=0}^{\lfloor \log_2 n \rfloor} \left\lfloor \frac{n}{2^j} \right\rfloor < \sum_{j=0}^{\infty} \frac{n}{2^j} = 2n$$

Thus, on average, each call to INCREMENT flips only $\frac{2n}{n} = 2$ bits, and so runs in constant time: $O(1)$. So the worst-case analysis has an amortized cost of $O(1)$.

The standard analysis is **not** wrong. It delivers an upper bound on the total time complexity which is much larger than it needs to be. This happens because the analysis fails to notice that not all bits are flipped at every single call of INCREMENT.

The second method for deriving amortized bounds is called the accounting method.

B. Accounting method

Assign an amortized cost to each operation. Amortized cost may be more or less than the actual cost for some operations. If the amortized cost is more than the actual cost of the operation, we assign the difference to specific objects of the data structure as credit. The credit can be used later to help pay for operations whose amortized cost is less than their actual cost. No negative credit is allowed. The total amortized cost should be an upper bound to the total actual cost. The basic concept with amortized analysis is that when we perform certain operations (usually $O(1)$ operations), we "put money in the bank" (credit) which is saved and used on "rainy days" (when needed).

EXAMPLE: Incrementing a binary counter

Recall: The running time is proportional to the number of bits flipped, which we use as amortized cost.

Amortized cost of flipping a bit from 0 to 1 is \$2. One of these dollars is spent flipping the bit from 0 to 1. The other dollar is stored away as credit until we need to reset the same bit back to zero. Note that we always have enough credit saved up to pay for the next increment

```
INCREMENT(A)
  begin
  j := 0
  while ((j < length(A)) ∧ (A[j] = 1))
       A[j] := 0                           cost: 0 !!
       j := j + 1
  if j < length(A)
       then A[j] := 1                      cost: 2
  end
```

Remark: The total amortized cost is always $\geq$ total worst-case cost.

X	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	amortized cost
1	0	0	0	0	1(1)	2
2	0	0	0	1(1)	0	2
3	0	0	0	1(1)	1(1)	2
4	0	0	1(1)	0	0	2
5	0	0	1(1)	0	1(1)	2
6	0	0	1(1)	1(1)	0	2
7	0	0	1(1)	1(1)	1(1)	2
8	0	1(1)	0	0	0	2
9	0	1(1)	0	0	1(1)	2
10	0	1(1)	0	1(1)	0	2
11	0	1(1)	0	1(1)	1(1)	2
12	0	1(1)	1(1)	0	0	2
13	0	1(1)	1(1)	0	1(1)	2
14	0	1(1)	1(1)	1(1)	0	2
15	0	1(1)	1(1)	1(1)	1(1)	2
16	1(1)	0	0	0	0	2
total amortized cost = 32						

Remark: (1) is the \$1 credit which is used to flip back the one to zero.

The amortized cost of an increment operation is \$2 (as is the case with the aggregate method). Also note that we always have enough credit to reset the flipped bits; credit is never negative.

The potential method is the third and last method for deriving amortized bounds.

C. The potential method

Rather than storing credit to represent prepaid work that can be spent later, we represent prepaid work as "potential energy" or just "potential". The term was borrowed from physics. The potential is associated with the entire data structure and not with specific parts of the data structure. Suppose that D_i represents the state or configuration of the data structure at some moment in time i. The state of D_i can be its size, its shape, the values it contains or other characteristics. The potential function Φ assigns a real number $\Phi(D_i)$ to D_i. The potential can be thought of as a measure of susceptibility of D_i to expensive operations. The more susceptible to expensive operations D_i is, the higher its potential $\Phi(D_i)$. Let c_i denote the actual cost of the i^{th} operation which transforms the data structure from state D_{i-1} to state D_i. The amortized cost $\overline{c_i}$ of the i^{th} operation is defined to be the sum of the actual cost and the change in potential:

$$\overline{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

The total amortized cost of the n operations is given by:

$$\begin{aligned}\sum_{i=1}^{n} \overline{c_i} &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

So the total amortized cost is the total actual cost plus the total change in potential. Note that if we can define a potential function Φ such that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^{n} \overline{c_i}$ will be an upper bound on the total actual cost $\sum_{i=1}^{n} c_i$.

Since we don't know a priori the number of operations performed, it is safer to require that $\Phi(D_i) \geq \Phi(D_0)$ for all i. So the goal is to define Φ such that $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$. Choosing a potential function that works well is a matter of trial-and-error and of course experience. Different potential functions will lead to different time bounds. One should try different possibilities with the data structure at hand.

Example: Incrementing a binary counter A

Define the potential of the counter after the i^{th} increment operation to be the number of ones in the counter. Denote the number of ones by b_i. Initially, all bits are equal to zero, $b_0 = 0$ and so $\Phi(D_0) = 0$. Moreover, $\Phi(D_i) > 0$ for all $i > 0$. Assume that the i^{th} call to increment resets to 0 t_i bits. The actual cost, c_i, is given by:

$$\begin{aligned} c_i &= \text{number of bits set to 1} + \text{number of bits reset to 0} \\ &\leq 1 + t_i \text{ since at most one bit is set to one}\end{aligned}$$

The number of ones in A after the i^{th} operation is performed is given by

$$\begin{aligned} b_i &= b_{i-1} - \begin{pmatrix}\text{number of ones} \\ \text{that are reset} \\ \text{to zero}\end{pmatrix} + \begin{pmatrix}\text{at most one} \\ \text{set to one}\end{pmatrix} \\ \Rightarrow b_i &\leq b_{i-1} - t_i + 1\end{aligned}$$

So

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &= b_i - b_{i-1} \\ &\leq b_{i-1} - t_i + 1 - b_{i-1} \\ &= 1 - t_i\end{aligned}$$

Therefore the amortized cost of the i^{th} operation is:

$$\begin{aligned}
\overline{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\leq (t_i + 1) + (1 - t_i) \\
&= 2
\end{aligned}$$

So here again the total amortized cost of a sequence of n INCREMENT operations is an upper bound on the total actual cost, and so the worst-case of n INCREMENT operations is $O(n)$.

Note that b_i is a measure of susceptibility of the state of the array A to the expensive operation of flipping bits. A state with a large b_i is more apt to undergo flips than a state in which we have a smaller value for b_i. □

In summary, define a potential function for the data structure such that $\Phi(D_0) = 0$ and $\Phi(D_i) > 0$ for all $i > 0$. The amortized cost of an operation is its actual cost plus the change in potential. Amortized analysis is a technique for bounding the running time of an algorithm. It is effective when analyzing an algorithm that performs a lot of cheap operations and occasionally performs expensive operations. The analysis allows us to balance the expensive-but-rare operations against their cheap-and-frequent counterparts.
We now resume our study of priority queues.

3.7.6 Fibonacci Heaps

Fibonacci Heaps were invented by Fredman and Tarjan in 1984. Priority queues can be very efficiently implemented by Fibonacci heaps. We shall use the method of amortized analysis with potential function to analyze Fibonacci heaps.

Recall:

- we quantify some aspect of the data structure by using a potential function. This measure is usually proportional to the amount by which the data structure is inefficiently arranged
- if operations are only expensive when the data structure is complicated and expensive operations can clean up the data structure, and it takes many cheap operations to increase the complexity of the data structure, then we can amortize the cost of the expensive operations over the cost of the many cheap operations to obtain a low average cost

Fibonacci Heaps are mergeable heaps. They maintain the minimum key in the heap at all times. Unlike Binomial Heaps, with Fibonacci heaps we can have one more copy of B_i for any i and trees are not necessarily binomial trees. A Fibonacci heap is a circular doubly linked list, with a pointer to the minimum key. The various trees, comprising the Fibonacci heap, are heap-ordered. Every node in a Fibonacci heap has a pointer to four other nodes:

- its parent
- its "next" sibling
- its "previous" sibling
- one of its children

The sibling pointers are used to join the roots together into a circular doubly linked list.

The advantages of a circular, doubly linked list are:

EXAMPLE:

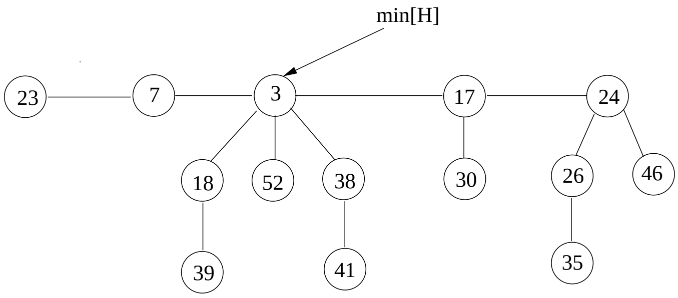


Figure 3.8: A high-level view of a Fibonacci Heap

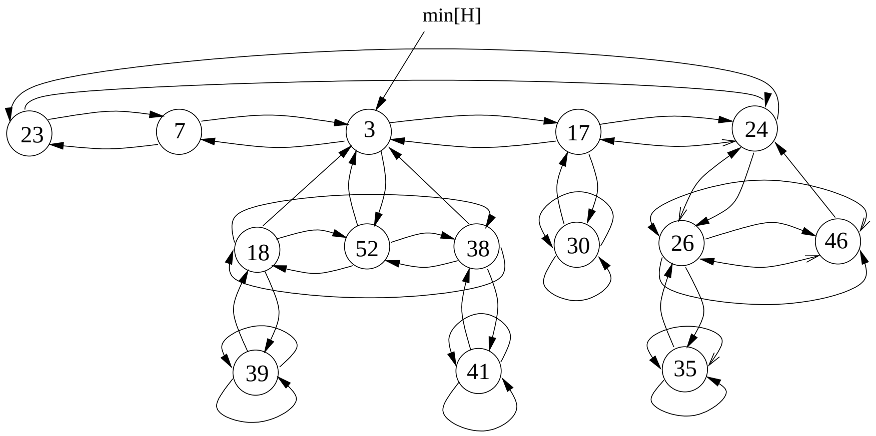


Figure 3.9: A detailed view of a Fibonacci Heap

□

1. removal of a node takes $O(1)$ time
2. concatenation of two circular, doubly linked lists into one circular, doubly linked list can be done in $O(1)$ time

The following table is a summary of the operations' running times for Binomial Heaps and Fibonacci Heaps.

procedure	Binomial Heap [worst-case]	Fibonacci Heap [amortized]
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$O(\log_2 n)$	$\Theta(1)$
MINIMUM	$O(\log_2 n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\log_2 n)$	$O(\log_2 n)$
UNION	$O(\log_2 n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\log_2 n)$	$\Theta(1)$
DELETE	$\Theta(\log_2 n)$	$O(\log_2 n)$

The number of items in the heap H at the time of the call to the procedure is denoted by n. As can be seen in the table, we use amortized analysis with Fibonacci heaps. In some operations, we shall mark nodes. If $t(H)$ denotes the number of trees in Fibonacci heap H and $m(H)$ denotes the number of marked nodes then the potential function of the Fibonacci heap is given by:

$$\Phi(H) = t(H) + 2m(H)$$

In the example, if nodes 18, 39 and 26 are marked then $\Phi(H) = 5 + 2 \cdot 3 = 11$. The potential of a set of Fibonacci heaps is the sum of the potentials of its constituent Fibonacci heaps. The most important idea with Fibonacci heap operations is to delay work as long as possible. When it is finally done, use the operation to simplify the data structure so that future work is easy.

Operations with fibonacci heaps

① Creating a new Fibonacci Heap
MAKE-FIB-HEAP creates an empty Fibonacci Heap H. $\Phi(H) = 0$ since $t(H) = m(H) = 0$. The amortized cost of MAKE-FIB-HEAP is $\Theta(1)$ (which is the actual cost)

② Inserting a node
FIB-HEAP-INSERT*(H, x)* inserts a node x in the Fibonacci Heap H by performing the following operations:

a) create a tree of type B_0 containing x

b) add this new tree to the list of roots of H

c) update *min[H]*, the minimum root pointer

d) increment $n[H]$ where $n[H]$ keeps track of the number of nodes of the Fibonacci Heap H

Let H' be the Fibonacci Heap obtained after performing
FIB-HEAP-INSERT*(H, 21)* (Figure 3.10).

$$\begin{aligned}
t(H') &= t(H) + 1 \\
m(H') &= m(H) \\
\Phi(H') - \Phi(H) &= [t(H') + 2m(H')] - [t(H) + 2m(H)] \\
&= [t(H) + 1 + 2m(H)] - [t(H) + 2m(H)] \\
&= 1
\end{aligned}$$

So the amortized cost = actual cost $+1 = O(1) + 1 = O(1)$.

③ Finding the minimum node
FIB-HEAP-MIN*(H)* returns the minimum key by returning the value of the node pointed to by *min[H]*. Actual time: $O(1)$, which is also equal to the amortized cost since the potential of H does not change.

④ Uniting two Fibonacci Heaps
FIB-HEAP-UNION(H_1, H_2) merges H_1 and H_2, destroying them in the process, and returns a new heap H, by performing the following operations:

a) combine the root lists of H_1 and H_2 to form a single list of heap H

b) update the minimum root pointer *min[H]*

c) set $n[H] = n[H_1] + n[H_2]$

EXAMPLE:

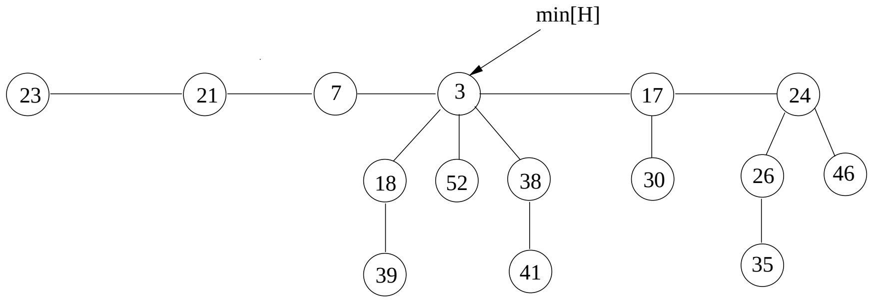


Figure 3.10: Fibonacci Heap H (Figure 3.8) after performing FIB-HEAP-INSERT*(H, 21)*

□

The change in potential is given by:

$$\begin{aligned}\Phi(H) - (\Phi(H_1) + \Phi(H_2)) &= [t(H) + 2m(H)] - [(t(H_1) + 2m(H_1)) \\ &\quad +(t(H_2) + 2m(H_2))]\end{aligned}$$

But $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$. So

$$\Phi(H) - (\Phi(H_1) + \Phi(H_2)) = 0$$

The amortized cost and the actual cost of FIB-HEAP-UNION are therefore equal. The cost of both is $O(1)$.

⑤ Extracting the minimum node
FIB-HEAP-EXTRACT-MIN*(H)* removes the minimum node and rearranges the heap. It is a complicated operation because this is where the delayed work of consolidating trees in the root list finally takes place. The main steps of FIB-HEAP-EXTRACT-MIN*(H)* are:

a) $z := min[H]$
b) delete z from the root list of H and add the children of z to the root list of H
c) call procedure CONSOLIDATE*(H)* which scans the root list and repeatedly links trees of equal degree, and finds the new minimum
d) decrement $n[H]$ by 1

Most of the work done in FIB-HEAP-EXTRACT-MIN*(H)* is performed by CONSOLIDATE*(H)*.

We want to reduce the number of trees in the Fibonacci Heap, i.e. we want to consolidate the root list of H. We link trees of equal degrees using procedure CONSOLIDATE*(H)*. But how do we know when we have trees of equal degree ?

The situation with binomial heaps is simple since the binomial trees are ordered in increasing order of degree. With Fibonacci Heaps, trees may occur in any order. A solution is to have the procedure CONSOLIDATE*(H)* use an array $A[1 \ldots D(n(H))]$ of the trees already scanned, indexed by degree.

CONSOLIDATE*(H)* initializes A to be empty. The procedure then tries to add a tree T of degree k in location $A[k]$. If that location is empty, the tree is

EXAMPLE: Extract the key with minimum value form the binomial heap H of Figure 3.10

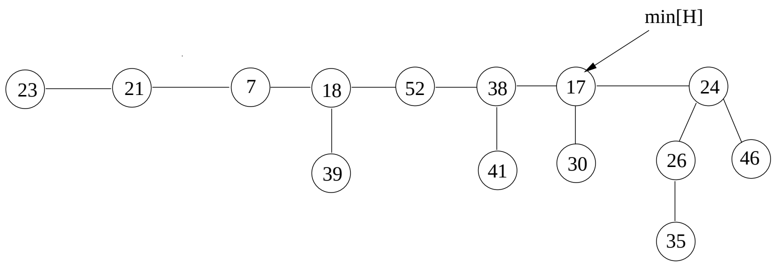


Figure 3.11: Heap H' (Figure 3.10) after performing steps a) and b) of FIB-HEAP-EXTRACT-MIN*(H)*. Note: $min[H]$ points to the right node of node 3

□

pointed to by $A[k]$ and CONSOLIDATE moves to another tree from the root list. But if location $A[k]$ is not empty, then we have another tree T' of degree k. CONSOLIDATE will then link T and T' by making the tree with the larger root a child of the tree with the smaller root and incrementing the smaller node's degree. Suppose that T'' is the resulting tree. CONSOLIDATE*(H)* will then recursively try to add T'' at location $A[k'']$, where k'' is the degree of T''. Note that at any given instant, if $A[k] = y$, then y is currently a root with degree k. At the end of the scan, when there are no more nodes in the root list of H, the trees are read out of A into a new root list with the new updated *min[H]*.

$D(n)$ is the maximum degree of any node in an n-node Fibonacci Heap. It can be shown that if only the mergeable heap operations (MAKE-HEAP, INSERT, MINIMUM, EXTRACT-MIN, UNION) are supported, then $D(n) = \lfloor \log_2 n \rfloor$.

EXAMPLE:

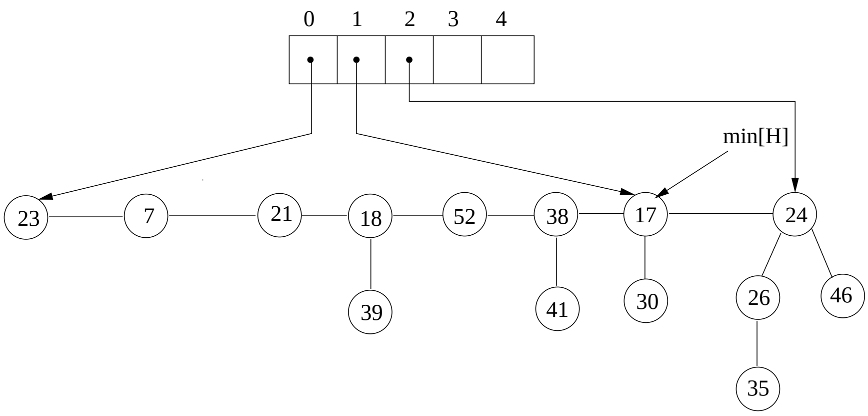
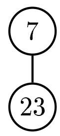


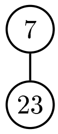
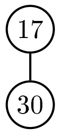
Figure 3.12:

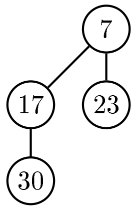
CONSOLIDATE*(H)* scans the roots 17, 24, and 23 (in that order) filling the array A of scanned trees as shown in Figure 3.12

Now the following steps take place:

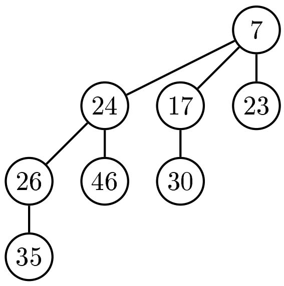
node 7

$A[0]$ is not empyt; so trees 7 and 23 get linked $\Rightarrow$ 

$A[1]$ is not empty; trees  and  get linked

$\Rightarrow$ 

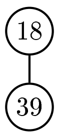
trees with roots at 7 and at 24 are linked

$\Rightarrow$ $A[3]$ points to 

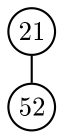
node 21

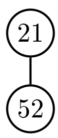
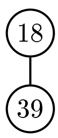
$A[0]$ points to 21

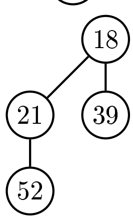
node 18

$A[1]$ points to 

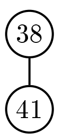
node 52

$A[0]$ is not empyt; link 52 and 21 $\Rightarrow$ 

$A[1]$ is not empty, link  and 

$\Rightarrow$ $A[2]$ points to 

node 38

$A[1]$ points to 

The trees are then read out of A with updated min. □

EXAMPLE:

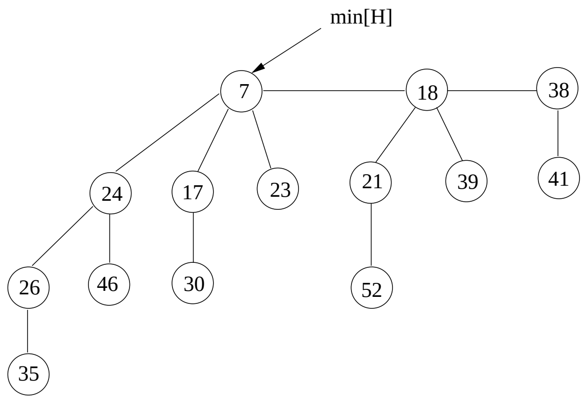


Figure 3.13: Result of FIB-HEAP-EXTRACT-MIN*(H)*

Let us analyze the work done to extract the minimum node. We shall show that the amortized cost of extracting the minimum node of an $n-$node Fibonacci Heap is $O(D(n))$. Let us compute the actual cost first. If the original heap had $t(H)$ roots then right prior to FIB-HEAP-EXTRACT-MIN*(H)*'s call to CONSOLIDATE*(H)*, the size of the root list is at most $D(n) + t(H) - 1$.
$D(n)$ is because of the minimum root's children, and
$t(H) - 1$ since we have already deleted the minimum root.

Since any root has at most $D(n)$ children, the operations of initializing the array A to null pointers and reading out the array at the end of CONSOLIDATE*(H)* take $O(D(n))$ time. Scanning the array checking for scanned trees in A by CONSOLIDATE*(H)* for every node in the root list is at most proportional to $D(n) + t(H)$. So the total actual work is

$$O(D(n) + t(H))$$

The potential before extracting the minimum is $t(H) + 2m(H)$, and the potential at the end of all operations is at most $(D(n) + 1) + 2m(H)$. Note that at most $D(n) + 1$ roots will remain after the execution of CONSOLIDATE*(H)*. Also neither procedure changes marked nodes. We know that

$$\text{amortized cost} \quad = \quad \text{actual cost } + \Phi(H') - \Phi(H)$$

So the amortized cost is at most:

$$\begin{aligned} & O(D(n) + t(H)) + [(D(n) + 1) + 2m(H)] - (t(H) + 2m(H)) = \\ = \quad & O(D(n)) + O(t(H)) - t(H) \\ = \quad & O(D(n)) \end{aligned}$$

Note that $O(t(H))$ and $t(H)$ cancel out because we can scale up the units of potential so as to balance the constant in $O(t(H))$.

6 Decreasing a key
the five operations seen so far are mergeable heap operations for which $m(H)$ plays no role in the potential function. Marking nodes is very important for

FIB-HEAP-DECREASE-KEY*(H, x, k)*. We say that x is marked if its mark is set to "true", and is unmarked if its mark is set to "false". A root is always unmarked. We mark x if it is not a root and it loses a child (i.e. one of its children is cut and put into the root list). We unmark x whenever it becomes a root. We shall make sure (later) that no marked node loses another child before it itself is cut (and so changes status to unmarked). Here we cut the node, change its key and insert it into the root list. If the parent of the node is unmarked, mark it. If the parent of the node is marked, cut it off also. Recursively perform the above until we get an unmarked node that we finally mark.

```
FIB-HEAP-DECREASE-KEY(H, x, k)
  begin
    assume k ≤ old key, otherwise send error message
    key[x] := k
    y := parent[x]
    if (y ≠ NIL) ∧ (key[x] ≤ key[y])
        then CUT(H, x, y)                          since heap-order
                                                   violated
             CASCADINGCUT(H, y)
    if key[x] < key[min[H]]
        then min[H] := x
  end
```

```
CUT(H, x, y)
  begin
    remove x from the child list of y
    decrement degree(y)
    add x to the root list of H
    parent[x] := NIL
    mark[x] := false
  end
```

When the heap-order is violated, many changes in the heap may happen (recall HEAPSORT). Marking nodes ensures that the running time functions of the operations are within the bounds we want. Consider following scenario for node x:

(a) x was a root

(b) then x was linked to another node

(c) then 2 children of x were removed by cuts

When x loses the second child, x is cut from its parent and joins the root list.

```
CASCADINGCUT(H, y)
  begin
  z := parent[y]
  if z ≠ NIL
      then if mark[y]=false
          then mark[y] :=true
          else CUT(H, y, z)
              CASCADINGCUT(H, z)
  end
```

We then set to *true* x's parent's mark, unless x's parent is a root. If the parent of x was already marked, then recursively cut the link between the parent and grandparent of x. These cuts may cascade upwards until reaching either a root or an unmarked node.

Note that marking prevents us from removing more than one child from a node that is not a root node. Whenever we try to remove a second child from a non-root node, we first convert the node into a root by cutting it away form its parent.

EXAMPLE:

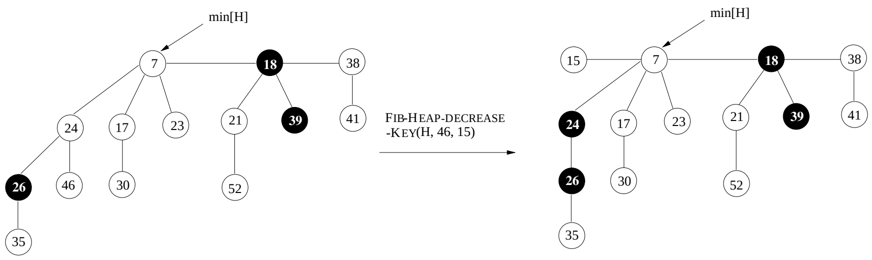


Figure 3.14: FIB-HEAP-DECREASE-KEY*(H, 46, 15)*

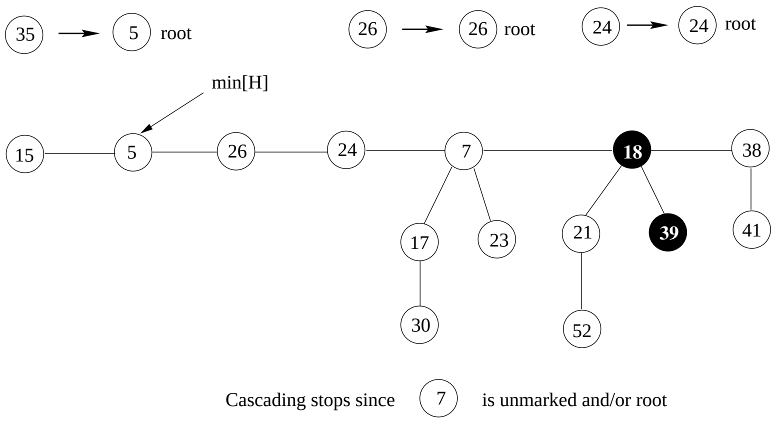


Figure 3.15: Based on the result of 3.14, we perform FIB-HEAP-DECREASE-KEY*(H, 35, 5)*

□

The amortized cost of FIB-HEAP-DECREASE-KEY is $O(1)$, as shown in the following:

(a) actual cost:
if we assume that FIB-HEAP-DECREASE-KEY calls CASCADINGCUT, which recursively calls itself c times, then the total actual time is $O(c)$. Note that c is bounded by the height of the tree

(b) change in potential
suppose that H is the original Fibonacci Heap (prior to the call to FIB-HEAP-DECREASE-KEY). Each recursive call of CASCADINGCUT, except for the last one (i.e. $c - 1$ calls), cuts a marked node and clears the

mark bit. If H' denotes the Fibonacci Heap after executing FIB-HEAP-DECREASE-KEY then:

$$t(H') = t(H) + \underbrace{(c-1)}_{\substack{\text{from} \\ \text{CASCADINGCUT}}} + \underbrace{1}_{\substack{\text{tree rooted} \\ \text{at } x}}$$

and

$$m(H') \leq m(H) - \underbrace{(c-1)}_{\substack{\text{unmarked by} \\ \text{CASCADINGCUT}}} + \underbrace{1}_{\substack{\text{last call to} \\ \text{CASCADINGCUT} \\ \text{may have marked} \\ \text{a node}}}$$

So $t(H') = t(H) + c$ and $m(H') \leq m(H) - c + 2$. Therefore

$$\begin{aligned}
\Phi(H') - \Phi(H) &= (t(H') + 2m(H')) - (t(H) + 2m(H)) \\
&= t(H') - t(H) + 2(m(H') - m(H)) \\
&\leq c + 2(-c + 2) \\
&= 4 - c
\end{aligned}$$

Remember: amortized cost = actual cost + change in potential
so we get:
amortized cost $\leq O(c) + 4 - c = O(1)$ by scaling up the units of potential to offset the constant hidden in $O(c)$.

In FIB-HEAP-DECREASE-KEY, a CASCADINGCUT which causes the node above it to cut because it was already marked, which causes the node above it to be cut since it too was already previously marked, etc ... can increase the actual cost (c in our analysis). We can pay for this cost by our potential function. Every cost we incur from having to update pointers due to a marked node that was cut is offset by the decrease in the potential function when that previously marked node is now left unmarked in the root list. Note that the potential function has $\underline{2}m(H)$ as one of its terms. When a marked node y is cut by CASCADINGCUT, its mark becomes *false*, so the potential is reduced by 2. One unit of potential pays for the cut and the clearing of the mark bit, and the other unit compensates for the unit increase in potential due to node y joining the root list.

7 Deleting a node
a node can be deleted from an $n-$node Fibonacci Heap in $O(D(n))$ amortized time.

```
FIB-HEAP-DELETE(H, x)
  begin
  FIB-HEAP-DECREASE-KEY(H, x, −∞)
  FIB-HEAP-EXTRACT-MIN(H)
  end
```

So x joins the root list and becomes the node with minimum value, and is then removed. Since the amortized time of FIB-HEAP-DELETE is the sum of the amortized time of FIB-HEAP-DECREASE-KEY (which is $O(1)$) and the amortized time of FIB-HEAP-EXTRACT-MIN (which is $O(D(n))$), we have $O(D(n))$ as amortized time for FIB-HEAP-DELETE. □

Bounding the maximum degree

We claimed (in the table) that extracting the minimum and deleting a node from Fibonacci Heaps each requires $O(\log_2 n)$ work. But the analysis we did was in terms of $D(n)$, the upper bound on the degree of any node of an $n-$node Fibonacci Heap. To achieve $O(\log_2 n)$ we have to show that $D(n)$ is $O(\log_2 n)$.

Note that when the Fibonacci Heap undergoes only mergeable-heap operations MAKE-HEAP, INSERT, MINIMUM, EXTRACT-MIN, UNION then $D(n) = \lfloor \log_2 n \rfloor$. Let $size(x)$ denote the number of nodes in the subtree rooted at x ($size(x)$ includes x also). We will show that $size(x)$ is exponential in $degree[x]$.

Result 1: *If $degree[x] = k$, and $y_1, y_2, \ldots, y_k$ is the order in which each y_i was linked to x (from the earliest to latest), then $degree[y_i] \geq i-2$ for $i = 2, 3, \ldots, k$.*
Proof: Since y_i is linked to x after the linking of $y_1, y_2, \ldots, y_{i-1}$, we have $degree[x] \geq i-1$ since $y_1, y_2, \ldots, y_{i-1}$ are linked to x before y_i. So by the time y_i is linked to x, we have $degree[x] \geq i-1$. But y_i is linked to x only if $degree[x] = degree[y_i]$. So $degree[y_i] \geq i-1$ at the time y_i is linked to x. Since the time it has been linked, y_i has lost at most one child (recall that if it loses two children, it is then cut from x and moved to the root list). So $degree[y_i] \geq i-2$. □

Note that if we had **only** binomial trees in the heap, then **result 1** would have given $degree[y_i] = i-1$ (instead of $degree[y_i] \geq i-2$).

Result 2: *If F_k denotes the k^{th} Fibonacci Number then $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$.*
Proof: Use induction on k:

$$\begin{aligned} 1 + \sum_{i=0}^{0} F_i &= 1 + F_0 \\ &= 1 + 0 = 1 = F_2 \end{aligned}$$

So the property holds for $k = 0$. Assume that $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$.

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &= F_k + (1 + \sum_{i=0}^{k-1} F_i) \text{ by induction hypothesis} \\ &= 1 + \sum_{i=0}^{k-1} F_i + F_k \\ &= 1 + \sum_{i=0}^{k} F_i \end{aligned}$$

So $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$ for all $k \geq 0$. □

It can be shown by induction that $F_{k+2} \geq \Phi^k$ where $\Phi = \frac{1+\sqrt{5}}{2} = 1.618$.

Result 3: *If x is a node in a Fibonacci Heap and $degree[x] = k$, then $size(x) \geq F_{k+2} \geq \Phi^k$, where $\Phi = \frac{1+\sqrt{5}}{2}$.*
Proof: Let s_k denote the minimum possible value of $size(z)$ over all nodes z such that $degree[z] = k$. In other words:

$$s_k = \min\{size(z) | degree(z) = k\}$$

Note that $s_0 = 1$, $s_1 = 2$, $s_2 = 3$, in general $s_k \leq size(x)$. Let $y_1, y_2, \ldots, y_k$ denote the children of x in the order in which they were linked to x.

$$size(x) \geq s_k \geq 2 + \sum_{i=2}^{k} s_{i-2}$$

But $s_k \geq F_{k+2}$. We show this by using induction on k. Recall that $F_0 = 0$, $F_1 = 1$, $F_2 = 1$, $F_3 = \ldots$. Therefore

$$\begin{aligned} s_0 &= 1 \geq F_2 \\ s_1 &= 2 \geq F_3 \end{aligned}$$

Assume $s_i \geq F_{i+2}$ for $i = 0, 1, \ldots, k-1$.

$$\begin{aligned} s_k &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\ &\geq 2 + \sum_{i=2}^{k} F_i \text{ by induction hypothesis} \\ &= 1 + 1 + \sum_{i=2}^{k} F_i \\ &= 1 + \sum_{i=0}^{k} F_i \\ &= F_{k+2} \text{ by result 2} \\ &\Rightarrow size(x) \geq s_k \geq F_{k+2} \geq \Phi^k \end{aligned}$$

□

Result 4: The maximum degree $D(n) = O(\log_2 n)$
Proof: Let x be any node in an $n-$node Fibonacci Heap. Suppose $degree[x] = k$.

$$\Rightarrow n \geq size(x) \geq \Phi^k$$

$$\Rightarrow k \leq \log_\Phi n$$

So $D(n)$ is $O(\log_2 n)$. □

Bibliography

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, 1990.