Greedy Strategy

A straightforward design strategy that can be applied to variety of problems

These problems have n inputs and a **feasible solution** is either a **subset of n** or a **permutation (ordering) of n** inputs that satisfies certain **constraints**.

There are many feasible solutions and getting one such feasible solution may not require much effort.

The feasible solution that **maximizes** or **minimizes** some **objective function** is called an **optimal solution**

The **greedy method** can be applied to solve **optimization problems**

There are two types of optimization problems

The solution is subset of n inputs where the order between the inputs is not important (subset paradigm)

The solution is a permutation of all n inputs or a subset of n inputs thus order is important (ordering paradigm)

Control abstraction for greedy algorithm for the subset paradigm

```
Algorithm greedy(a,n)
{ solution =φ
for i=1 to n do
   { x=select(A);
     if feasible(solution , x) then
          Solution = Union(solution, x)
   }
return solution
}
```

Knapsack problem (fractional)

There are n objects and a knapsack or bag.

Object i has weight w_i and the knapsack has capacity m.

If a fraction x_i of, $0 \le x_i \le 1$, of object I is placed into the knapsack then a profit $p_i x_i$ is earned

Since the knapsack capacity is m, the total weight of all chosen objects can be at most m

$$\sum_{1 \le i \le n} w_i x_i \le m \qquad 0 \le x_i \le 1$$

The objective is to obtain a filling of the knapsack that maximizes the total profit earned.

$$\text{maximize} \quad \sum_{1 \le i \le n} p_i x_i$$

A feasible solution is any set$(x_1, x_2, \ldots x_n)$ satisfying the constraint $\sum_{1 \le i \le n} w_i x_i \le m$ and $0 \le x_i \le 1$

An optimal solution is for which $\sum_{1 \le i \le n}$ pi xi is maximized

Example

n=3 m=20 (w1,w2,w3)=(18,15,10) (p1,p2,p3)=(25,24,15)

There are many feasible solutions

	(x1 x2 x3)	Total weight	Total profit
1	(1/2, 1/3, 1/4)	16.5	24.25
2	(1, 2/15, 0)	20	28.2
3	(0, 2/3, 0)	20	31
4	(0, 1, 1/2)	20	31.5

In case sum of all the weights is ≤ m then xi=1 , 1≤ i ≤n, is an optimal solution

In case sum of all the weights is > m, then all optimal solutions will fill the knapsack exactly.

There are several Greedy approaches

1. Chose the object which yields highest profit-

Selection-Select objects in non-increasing order of profit

Feasibility-chose xi to ensure feasibility - as 1 if wi ≤ remaining capacity r else as the ratio of r/wi

```
Algorithm GreedyKnapsackprofit(P,W,m,n,X)
{ // arrays ordered in non-increasing order of profits
for i=1 to n do  X[i]=0
  r=m ; p=0 ; w=0;
for i=1 to n
 { if W[i] ≤ r then
      {  X[i]=1;
          p=p+P[i] ;
          r=r-W[i] ;
         w=w+W[i]  }
   else { x[i]=r/W[i] ;
          p=p+ P[i]* r/W[i];
         w=w + r;
         return p}
} return p }
```

r=20 p=0 w=0

18 < 20

X[1]=1

p=25 r=2 w = 18

15 > 2

X[2]= 2/15

p=25+(2/15)*24=28.2 w=20

This greedy strategy gives a suboptimal solution. Though profit increases at every step, knapsack capacity is used up rapidly

2. Chose the object which takes up the least capacity of knapsack –one with smallest weight

By being greedy about capacity we can earn more profits

Selection- select objects in non-decreasing order by weight.

Feasibility -chose xi as 1 if wi ≤ remaining capacity else chose xi as the ratio of remaining capacity by wi

```
Algorithm GreedyKnapsackweight(P,W,m,n,X)
{ // arrays ordered in non-decreasing order of weights
……………..// same as GreedyKnapsackprofit }
```

r=20 p=0 w=0

10 < 20

X[1]=1 p=15 r=10 w = 10

15 > 10

X[2]= 10/15 p=15+(10/15)*24=31 w=20

This greedy strategy also gives a suboptimal solution

Though capacities are used slowly, profits are not increasing rapidly

3. Chose the object having maximum profit perunit of capacity

Selection-select objects in non-increasing order by the ratio pi/wi.

```
Algorithm GreedyKnapsackratio(P,W,m,n,X)
{ // arrays ordered in non-increasing order of pi/wi
……………..// same as GreedyKnapsackprofit }
```

(24/15,15/10,25/18)=(1.6,1.5,1.39)

r=20 p=0 w=0

15 < 20

X[1]=1 p=24 r=5 w = 15

10 > 5

X[2]= 5/10 p=24+(5/10)*15=31.5 w=20

It can be proved that this greedy strategy gives an optimal solution

0/1 Knapsack problem

The additional constraint that xi=1 or xi=0, that is and object is either included or not included into the knapsack, fraction is not allowed

N=4 m =20 wi(16, 12 , 8, 6) pi(32, 20, 14, 9)

1 greedy strategy ordered according to profit

1. (1,0,0,0) total weight=16 total profit=32

2 greedy strategy ordered according to weight

2. (0, 0,1,1) total weight=14 total profit=23

3 greedy strategy ordered according to pi/wi

(32/16,14/8,20/12, 9/6)=(2, 1.75,1.67,1.5)

3. (1,0,0,0) total weight=16 total profit =32

Consider the feasible solution

4 (0,1,1,0) total weight=20 total profit =34

None of the greedy strategies give an optimal solution in case of 0/1 knapsack problem

Job Sequencing with deadlines

There are n jobs . With every job is associated a deadline di > 0and a profit pi.

The profit is earned if and only if job is completed before deadline

To complete a job, the job is to be processed on a machine for one unit of time. Only one machine is available for processing jobs

A feasible solution is a subset J of jobs such that each job in this subset can be completed by its deadline

There are many feasible solutions . All singleton sets are feasible solutions

Value of a feasible solution J is the sum of the profits of the jobs in J.

An optimal solution is a feasible solution with maximum value

Example

N=4 (p1,p2,p3,p4)=(50,10,15,27) and (d1,d2,d3,d4)=(2,1,2,1)

	Subset	**Processing sequence**	**Value**
1	**{1,2}**	**2, 1**	**60**
2	{1,3}	1,3 or 3, 1	65
3	{1,4}	4,1	77
4	{2,4}	not feasible	
5	{3,4}	4,3	42
6	{1,2,3}	not feasible	
7	{1}	1	50

The objective of Greedy algorithm is to increase profit

Selection- select the in non-increasing order of profits

Feasibility-one way is to try all possible permutations of selected jobs to get one that does not violate deadlines -which is time consuming

Feasibility can be checked by using only one permutation of jobs –ordered in non-decreasing order of deadlines

If this permutation violates deadlines, then no other permutation can satisfy deadlines

The greedy algorithm thus obtained always gives an optimal solution

```
Algorithm JobScheduling(D, J, n)
    // jobs are in non-increasing order of profits
    {D[0]=J[0]=0;// sentinel for insertion
    J[1]=1; // include the first job
    k=1; // no of jobs included
    for i=2 to n do
    { r=k; // search for a position for ith job in 1 to r
    //if the deadline of job at rth position equals r it cannot be
  //pushed down also for insertion deadline of ith job //should
  be less than job at rth position
    While D[J[r]] >D[i] and  D[j[r]] ≠r   do r = r-1
//the job is to be now inserted at r+1 th position only if its
//deadline is greater than r and greater than the Job at r
if( D[J[r]]<=D[i] and D[i] >r then
{ // insert job by pushing down jobs to create space for
    //insertion
   for q=k downto r+1(**q = k; q>=r+1 ;q--) do J[q+1]=J[q]
      J[r+1]= i
         k = k+1
}
} return k
}
```

The worst case computing time of the algorithm is $\theta(n^2)$

Better computing time can be obtained by postponing processing of job as much as possible

Divide the time into b slots where b=min { n, max{ di }}

For job i find the slot closest to its deadline that is free and allot it, if none such slots are free job is rejected

Example: N=5 (p1,p2,p3,p4,p5)= (20,15,10,5,1) and
(d1,d2,d3,d4,d5)=(2,,2,1,3,3)

There are initially three slots(3= min(5, max{di}))

[0,1], [1,2] and [2,3] and all are free

The jobs are ordered in non-increasing order of profits

Selected job	deadline	slot allocation	action taken	profit
1	2	[1,2] is free	assigned	20
2	2	[1,2] not free		
		[0,1] is free	assigned	15
3	1	[0,1] not free		
		no more slots	rejected	0
4	3	[2,3] is free	assigned	5
5	3	[2,3] not free		
		[1,2] not free		
		[0,1] not free		
		no more slots	rejected	0

Total profit earned =40

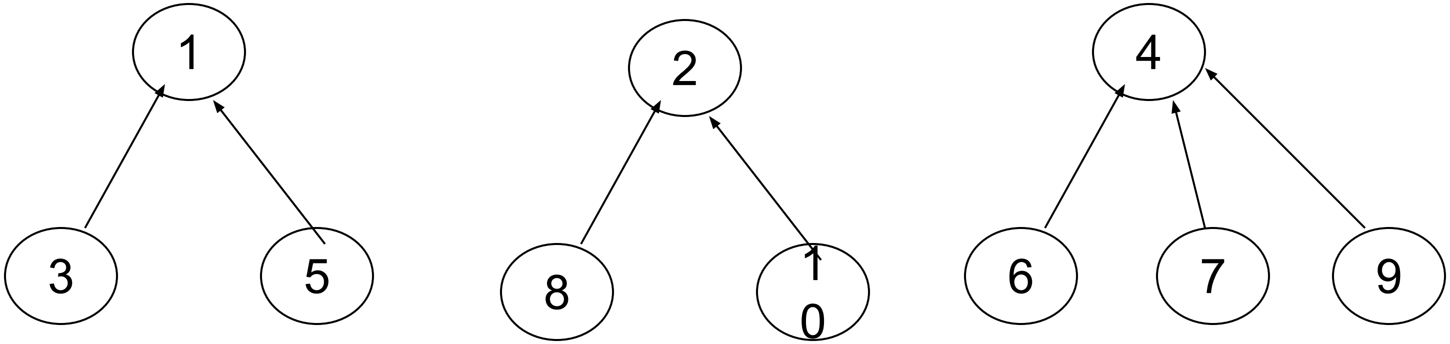
The algorithm can be implemented by using an efficient data structure disjoint set with unions

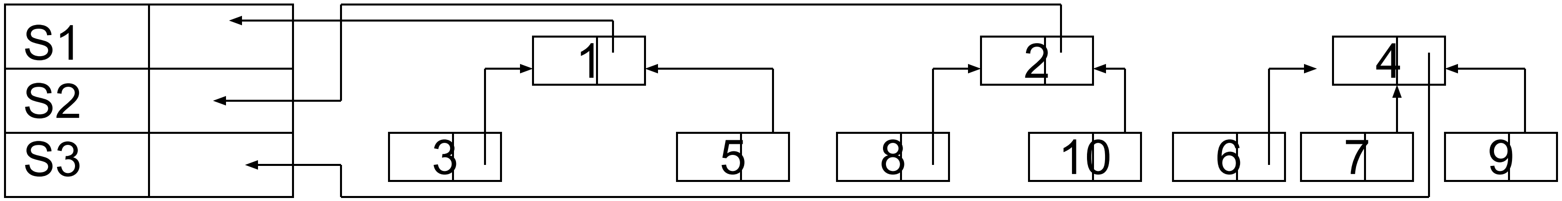
Data structure Disjoint set union

Sets of elements which are pair wise disjoint, i.e, if Si and Sj are two sets, then there is no element that is both in Si and Sj.

Ex S1={1,3,5} S2={2,8,10} S3={4,6,7,9}

Each set can be represented as a tree with each node stores a link to its parent. One of the nodes can be a parent

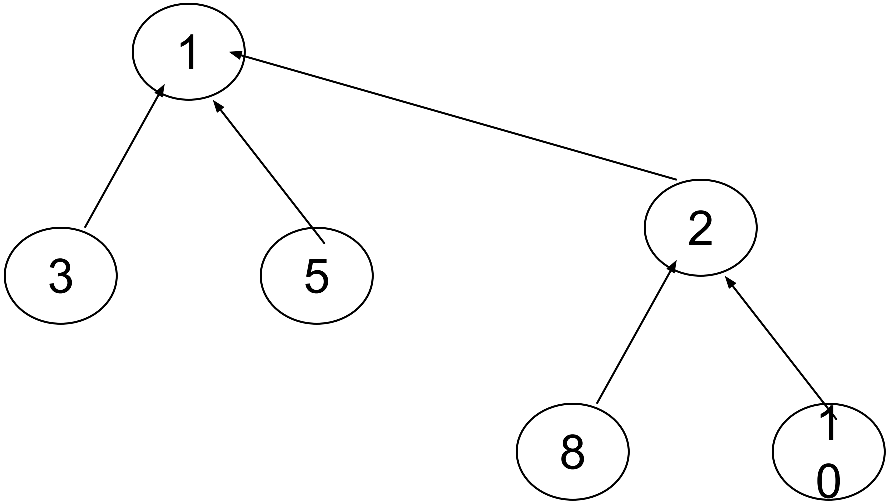




The operations to be performed on disjoint set

1. Disjoint set union – If Si and Sj are two disjoint sets , then the union Si U Sj is the set of all elements in Si and Sj

Ex S1={1,3,5} S2= { 2,8,10} S1Us2={1,3,5,2,8,10}



Make one of the trees a subtree of the other

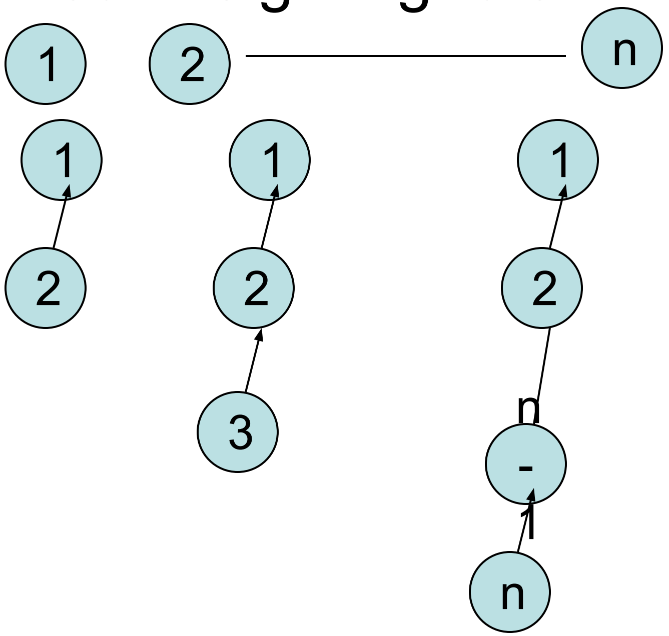
Set the parent field of one of the roots to other root

2. Find - Given the element i find the set containing i.

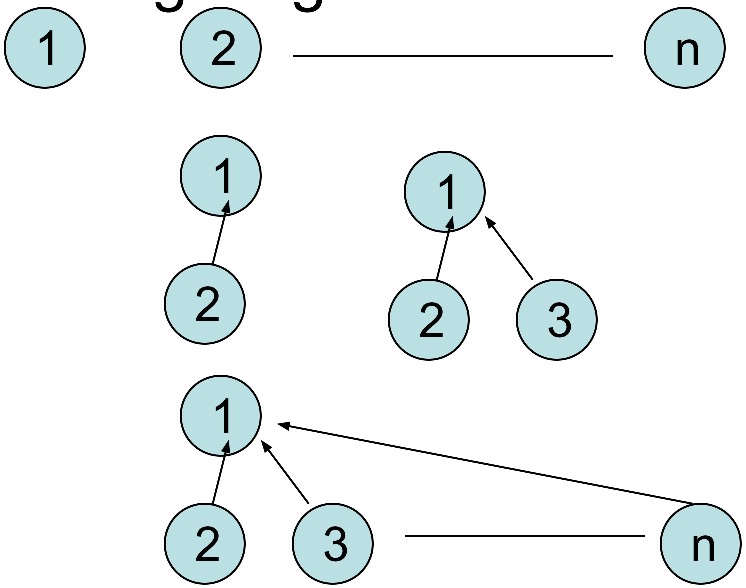
The performance of find and union can be improved by avoiding the creation of degenerate trees.

Weighting rule-If the number of nodes in the tree with root i is less than the number in the tree with root j, then make j the parent of i otherwise make i the parent of j. It is applied during union operation

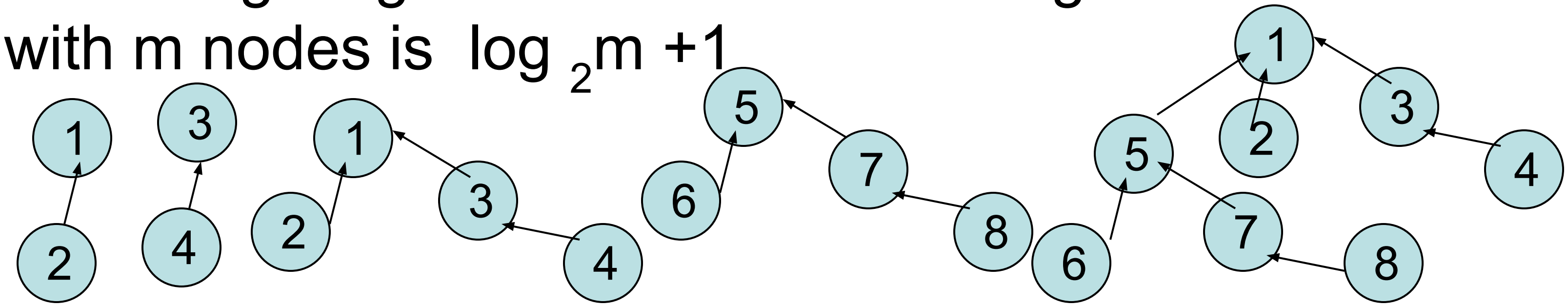
Without weighting rule



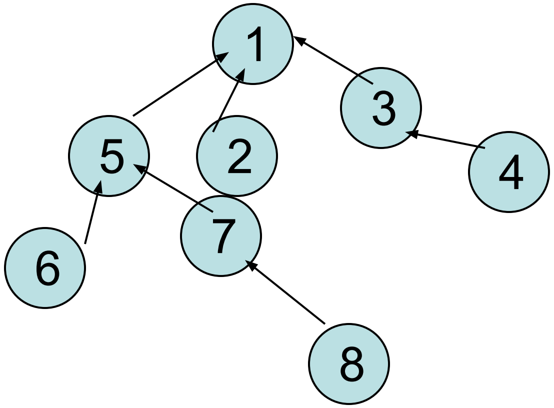
With weighting rule



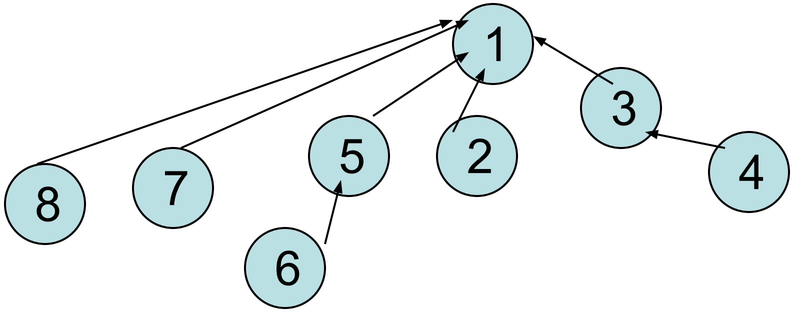
With weighting rule the worst case height of the tree with m nodes is $\log_2 m + 1$



Collapsing rule-If j is a node on the path from i to its root and p[i] (parent of i) is not equal to root[i], then set p[j] to root[i]-Collapsing rule is applied during find operation



First Find(8) requires going up three parent links and then two links are reset



Remaining Find(8)s will require only going up one link

The weighted unions and collapsing almost double the time required for individual find but reduce the time for subsequent finds.

- The effect of a collapsing find operation. After the find, all the nodes along the search path are attached directly to the root.
- That is, they have had their depths decreased to one. As a side-effect, any node which is in the sub tree of a node along the search path may have its depth decreased by the collapsing find operation.
- The depth of a node is never increased by the find operation.

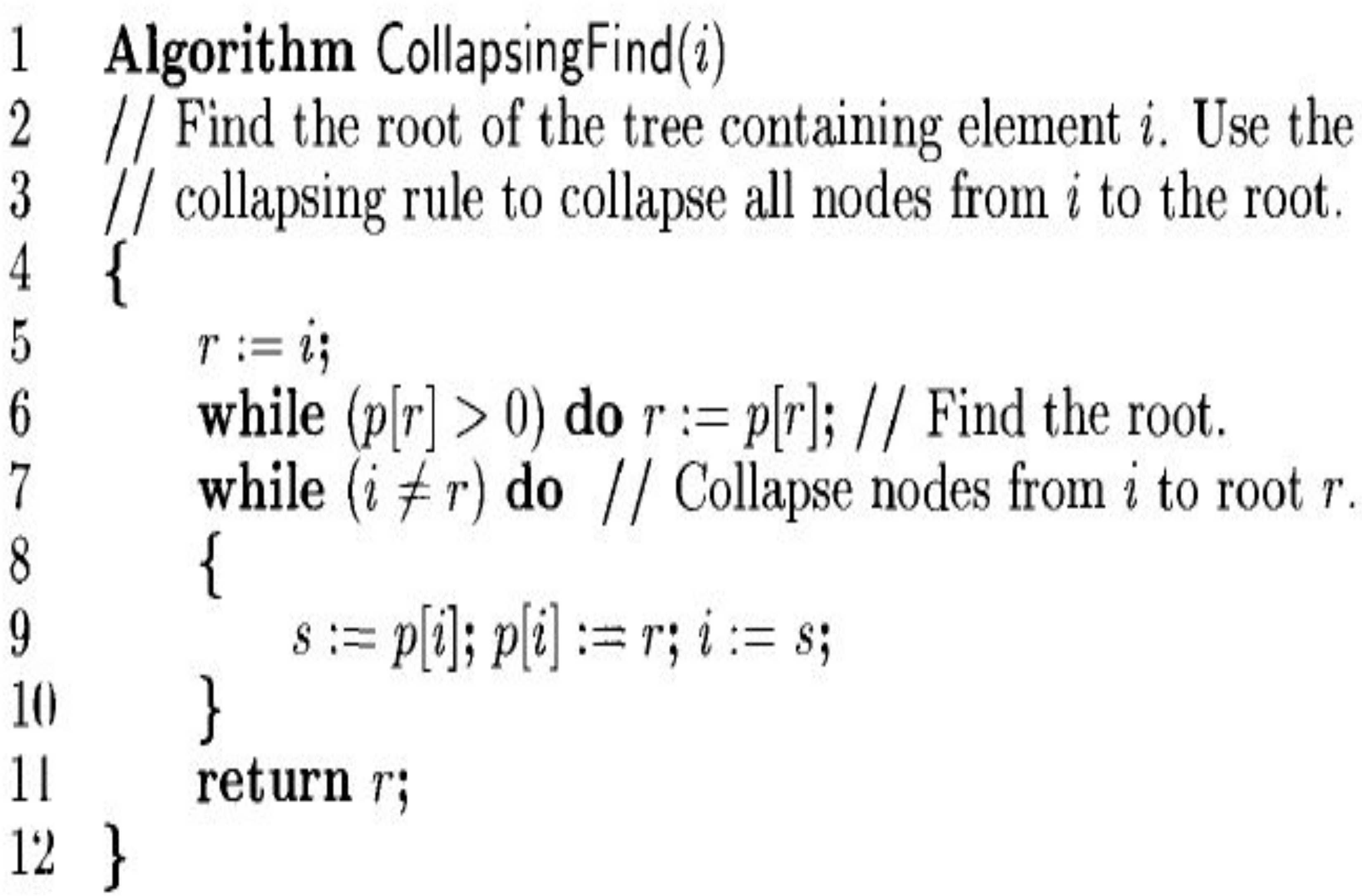
The time complexity of weighted union and collapsing finds is much better than $\log_2 n$

Disjoint set with weighted unions and collapsing finds is a good data structure

```
Algorithm FastJobScheduling(D, J, n, b)
{for i=0 to b do p[i]=i  // initialize
k=0;// the number of jobs selected
for i=1 to n do
    {q = collapsingFind(min(n, D[i]))
    if p[q] = 0 then write(“rejected”)// if it is root
      else {
        k=k+1; J[k]=i //select job i
              m=CollapsingFind(p[q]-1)
       Weighted Union(m, q)
          p[q]=p[m]; //q may be new root
       }
   }//end of for
}
```

```
Algorithm FJS($d, n, b, j$)
// Find an optimal solution $J[1:k]$. It is assumed that
// $p[1] \geq p[2] \geq \cdots \geq p[n]$ and that $b = \min\{n, \max_i(d[i])\}$.
{
    // Initially there are $b+1$ single node trees.
    for $i := 0$ to $b$ do $f[i] := i$;
    $k := 0$; // Initialize.
    for $i := 1$ to $n$ do
    { // Use greedy rule.
        $q :=$ CollapsingFind($\min(n, d[i])$);
        if $(f[q] \neq 0)$ then
        {
            $k := k+1$; $J[k] := i$; // Select job $i$.
            $m :=$ CollapsingFind($f[q]-1$);
            WeightedUnion($m, q$);
            $f[q] := f[m]$; // $q$ may be new root.
        }
    }
}
```

```
Algorithm WeightedUnion($i, j$)
// Union sets with roots $i$ and $j$, $i \neq j$, using the
// weighting rule. $p[i] = -count[i]$ and $p[j] = -count[j]$.
{
    $temp := p[i] + p[j]$;
    if ($p[i] > p[j]$) then
    { // $i$ has fewer nodes.
        $p[i] := j$; $p[j] := temp$;
    }
    else
    { // $j$ has fewer or equal nodes.
        $p[j] := i$; $p[i] := temp$;
    }
}
```



```
**Algorithm** CollapsingFind($i$)
// Find the root of the tree containing element $i$. Use the
// collapsing rule to collapse all nodes from $i$ to the root.
{
    $r := i$;
    **while** $(p[r] > 0)$ **do** $r := p[r]$; // Find the root.
    **while** $(i \neq r)$ **do**  // Collapse nodes from $i$ to root $r$.
    {
        $s := p[i]$; $p[i] := r$; $i := s$;
    }
    **return** $r$;
}
```

Optimal Merge patterns

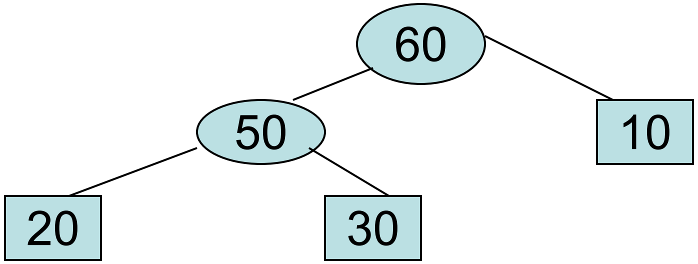
Two sorted files containing m and n records can be merged together to get a sorted file in O(m+n)

If there are n files to be merged, there are several possible orders in which merging can take place with different associated costs

Ex n =3 files of size 20 , 30 and 10

1. [[1 2] 3] 20+30=50 50+10=60 Total 110
2. [1 [2 3]] 30+10=40 20 +40=60 Total 100
3. [[1 3] 2] 20+10=30 30 +30=60 Total 90

The above merge patterns are two way merge patterns and can be represented in the form of binary merge trees



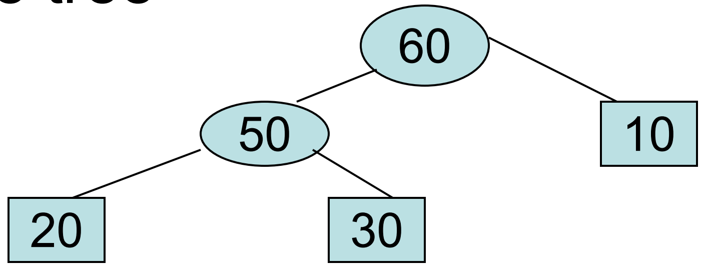
The leaf nodes represent files and are called external nodes

The internal nodes represent files obtained by merging files represented by its children O(m+n) . The number in node is the length of the file

The root represents the final file obtained after merging

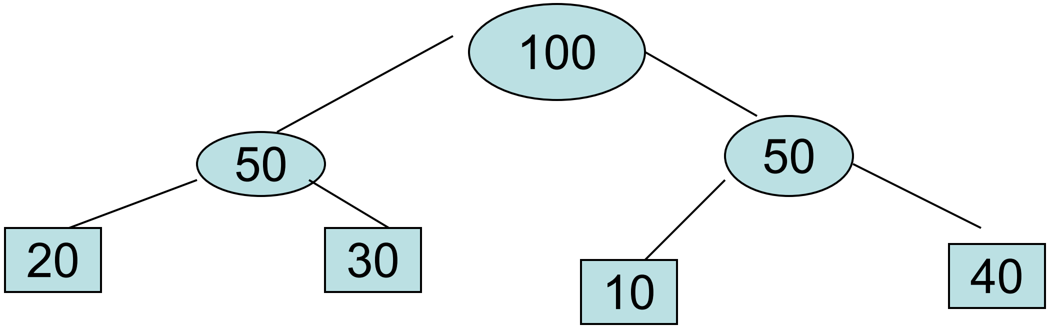
If di is the distance from root to an external node of length qi then the total number of moves for binary merge tree is ∑ diqi .

This sum is called the weighted external path length of the tree



2(20)+2(30)+1(10)=110

2(20)+2(30)+2(10)+2(40)=200



The optimal two way merge pattern corresponds to a binary merge tree with minimum weighted external path length

The greedy approach is at every stage select the two smallest files and merge them. The new file obtained is a used in subsequent merges

Each file is a tree node with left child , right child and weight which is the file length

The tree nodes are maintained as a list with two operations

Least(list) –returns the smallest element

Insert(list, t)- inserts the tree node in the list

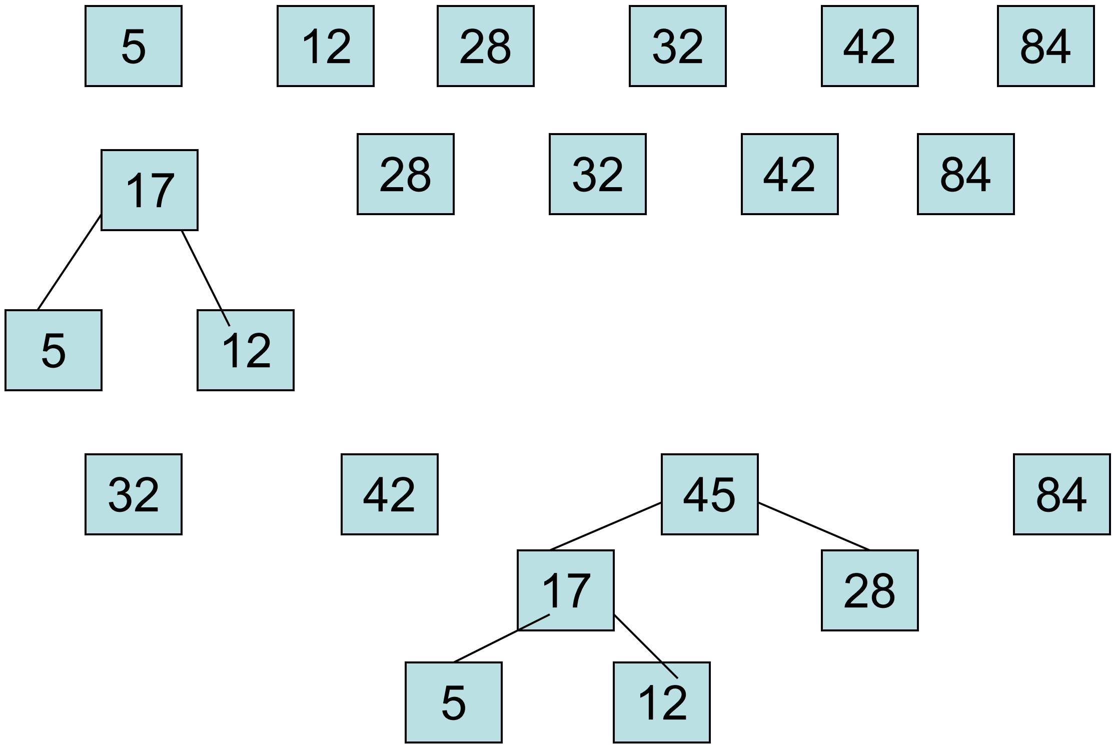
```
Algorithm OptimalMergePattern(list, n)
{ for i=1 to n-1 do
    {
    new->lchild=Least(list) ;
   new->rchild=Least(list) ;
   new->weight=((new->lchild)->weight) + ((new->rchild)->weight) ;
        Insert(list, new);
    }
return Least(list) ;
}
```

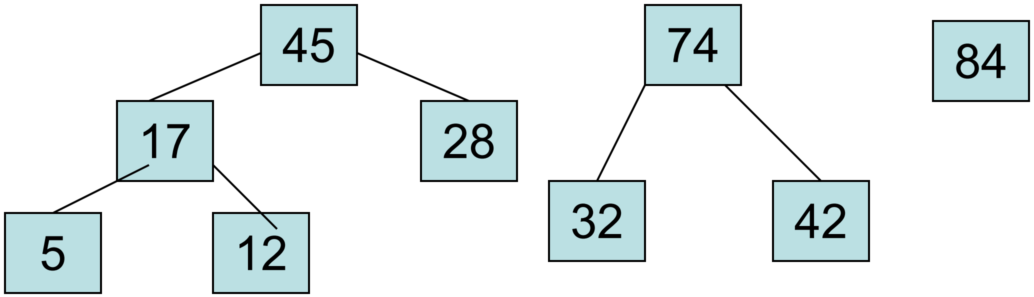
Example

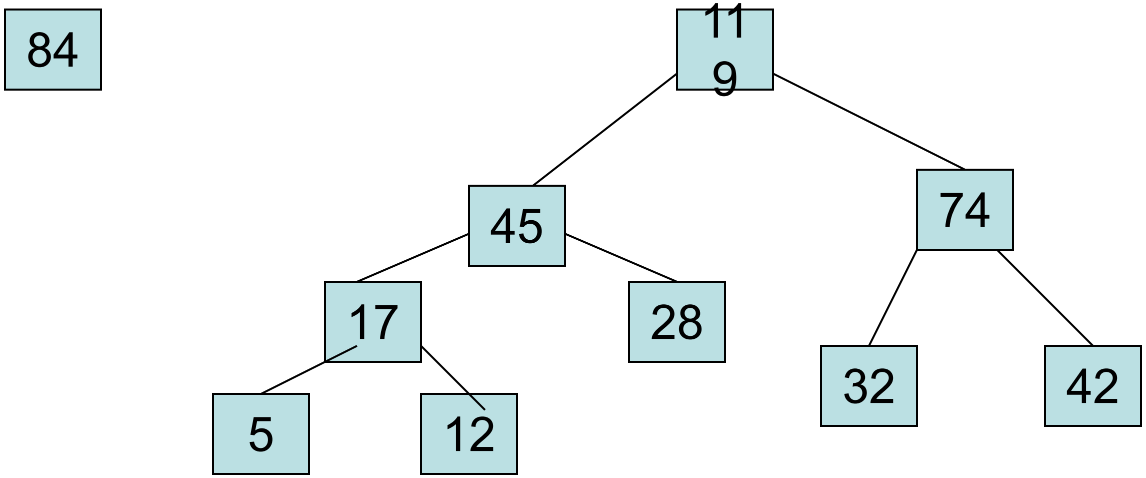
N=6 file sizes are

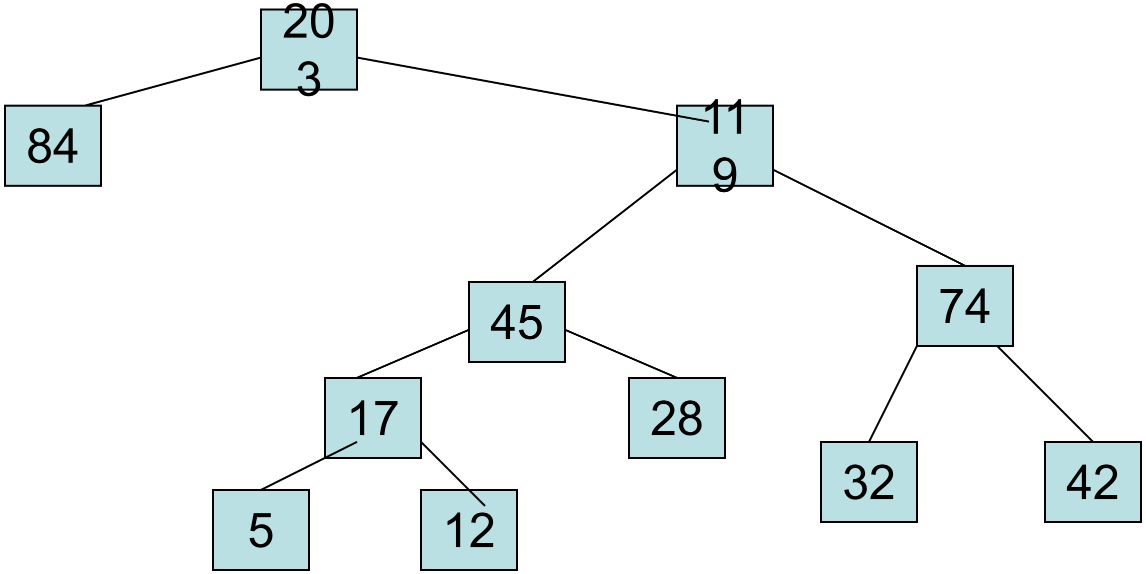
28, 32, 12, 5, 84, 42

List contains




45
17
28
5
12
74
32
42
84


84
11
9
45
74
17
28
32
42
5
12



If list is kept in non-decreasing order according to weight

Least(list) requires only O(1) time while insert(list,t) will take O(n) time . The algorithm will be of O(n^2)

The list can be maintained as priority queue with Least as delete operation and usual insert operation on priority queues.

If priority queues are maintained as heaps then the both operations are of O(logn)

The running time of algorithm will be O(nlogn)

Huffman Codes

Huffman codes are widely used and very effective technique for compressing data

Coding is representing each character by a binary string called codeword

Two types of codes

1. Variable length codes
2. Fixed length codes

A variable length code can considerably reduce the message size by giving frequent characters short codeword and infrequent characters long codeword

	a	b	c	d
Frequency	100	20	30	5
Fixed length encoding	00	01	10	11
Variable length encoding	0	10	110	111

Message aabbbaaaacaacad

Fixed length encoding 15x2=30

Variable length 9+6+9 =24

Prefix codes are codes in which no codeword is also a prefix of some other codeword

Prefix codes are used because they simplify encoding (compressions) as well as decoding

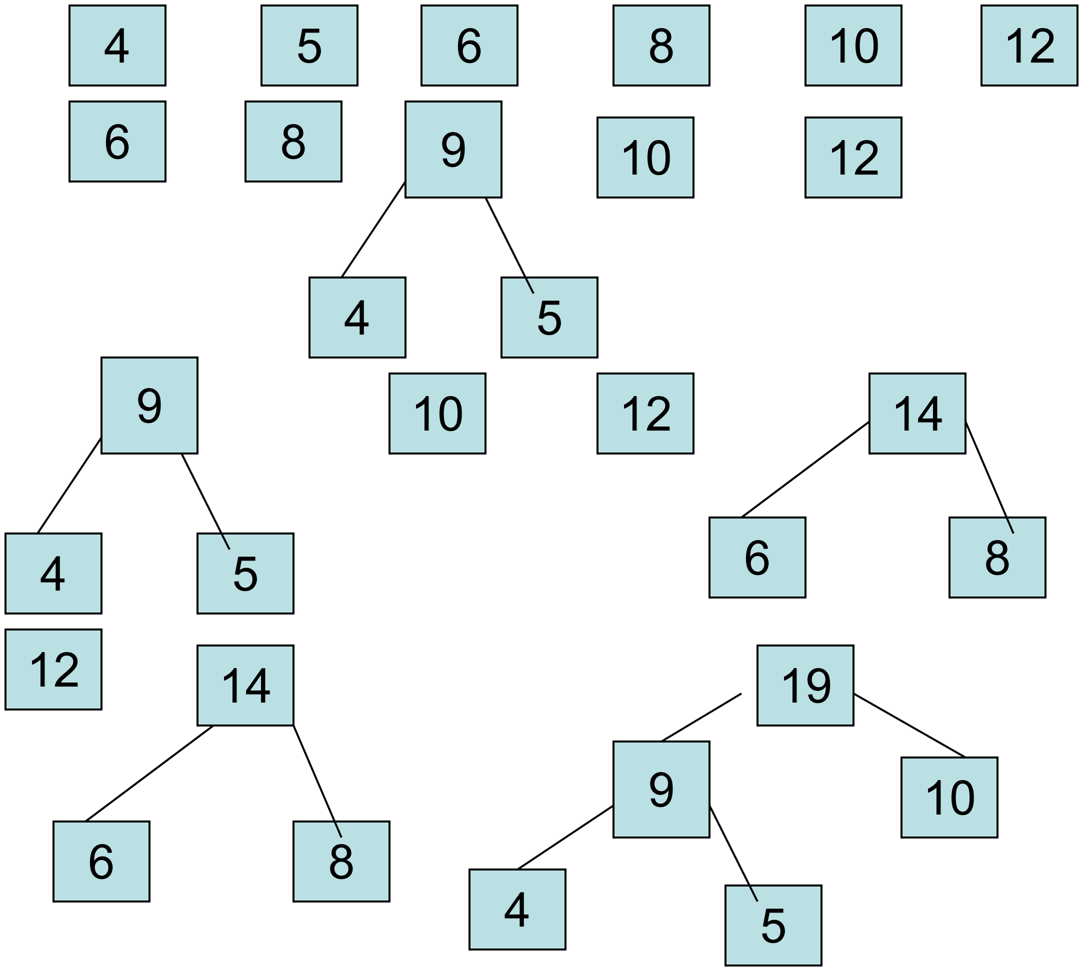
Huffman used a greedy approach that uses table of frequency of occurrence of each character to build an optimal way of representing each character as a binary string(codeword)

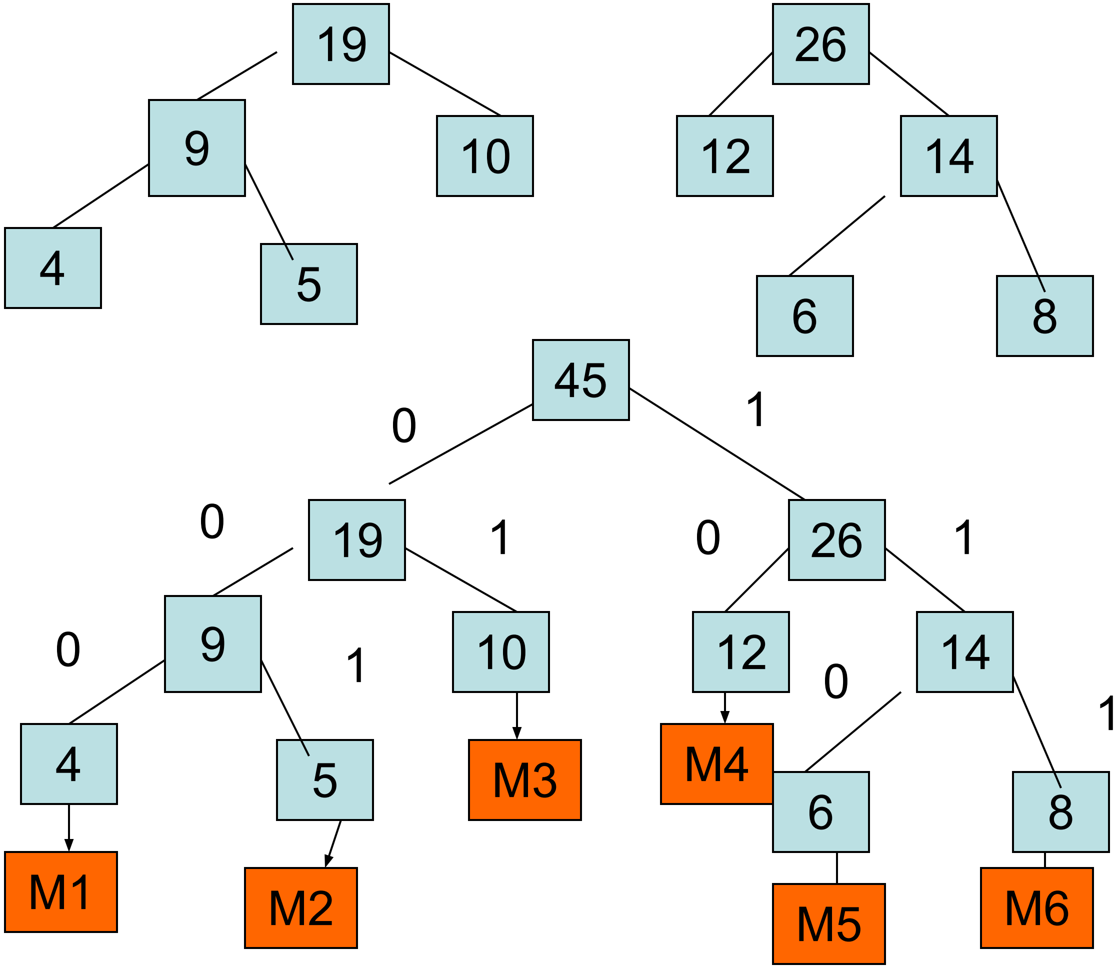
The decoding process needs a convenient way of representing the prefix codes

A binary tree is constructed using merge operation same as in optimal merge patterns whose leaves are given character frequencies.

The binary codeword for a character is the path from the root to that character where 0 means go to the left and 1 means go to right

Ex n=6 Messages with frequencies 4 ,5 ,6 ,8, 10, 12





M1= 000

M2=001

M3=01

M4= 10

M5 = 110

M6 = 111

- Obtain set of Huffman codes for the messages (m1,m2,m3,m4,m5,m6,m7) with relative frequencies(f1,f2,f3,f4,f5,f6,f7)=(1,1,2,3,5,8,13).

Optimal storage on tapes

There are n programs that are to be stored on a tape of length l. Associated with each program is a length l_i, $1 \leq i \leq n$. all programs can be stored if sum of the lengths of the programs is at most l.

If the programs are stored in order I= $i_1, i_2 \ldots . . i_n$, the time t_j needed to retrieve program i_j is proportional to $\sum_{1 \leq k \leq j} l_{ik}$ $(l_1 + l_2 + \ldots + l_j)$

If all programs are retrieved equally often , then the expected or mean retrieval time MRT is $1/n \sum_{1 \leq j \leq n} t_j$

The optimal solution is looking for a permutation of n programs that minimizes MRT

The greedy method requires to store the programs in non-decreasing order of their lengths

The greedy algorithm reduces to an efficient sorting algorithm