SOME DATA STRUCTURES

The use of well-chosen data structures is often a crucial factor in the design of efficient algorithms. Students are expected to have some basic knowledge on: arrays - records - and the various structured data types obtained using pointers. They are also expected to have some familiarity with the mathematical notions of directed and undirected graphs, which we will review shortly

5.1 Arrays, Stacks and Queues:

An array is a data structure consisting of a fixed number of items of the same type. On a machine they are stored (usually) in contiguous storage cells.

tab: array [1..50] of integers

Here tab is an array of 50 integers indexed from 1 to 50, tab[6] refers to the 6th item of the array. (It is natural to think of the items to be arranged from left to right.)

An interesting (to us) property of arrays is that we can calculate the address of any item in constant time.

Example: Assume array tab starts at address 5000, and each integer variable occupy 4 bytes of storage each, then the address of the item with index k is $4996 + 4.k$. This time is obviously bounded by a constant. It follows that the time required to read the value of an item, or to change such a value is in $O(1)$: in other words we treat such operations as elementary.

On the other hand, any operation that involve all the items of an array will tend to take longer as the size increases. Suppose we deal with an array of size n (n: items). An operation,

like initialization or finding the largest element, takes a time proportional to the number of items to be examined. It takes $\theta(n)$. Another common situation is where we want to keep the values of successive items of the array in order (numerical, alphabetic..). Now, whenever we decide to insert a new value we have to open up a "space" in the correct position, by shifting either lower or higher values one position the left or right. In the worst case we have to move $\frac{n}{2}$ items. Similarly deleting an element may require us to move all or most of the remaining items in the array. Such operations take a time in $\theta(n)$.

A one-dimensional array provides an efficient way to implement the data structure called a Stack. Here, items are added and removed on a last-in-first-out basis (LIFO). Index in this case run from 1 to maximum required size of the stack, where a counter is used. To set the stack empty, counter is set to zero; to add an item the counter is incremented, and the new item is written in stack[counter], to remove an item the value of stack[counter] is read out, and the counter is decremented. Adding an element to the Stack is called a "PUSH", removing is "POP".

Data Structure called "queue" can also be implemented efficiently in a one-dimensional array. Here, items are added and removed on a first-in-first-out (FIFO) basis. Adding an item is called "ENQUEUE" operation, while removing one is called "DEQUEUE".

One major disadvantage with stacks and queues: Space allocation has to be envisaged at the outset for the maximum number of items. If ever the space is not sufficient, it is difficult to add more, while if too much space is allocated, waste results.

Arrays with 2 or more indexes, can be declared in the same way as one dimensional arrays. For instance,

matrix : array [1..20, 1..20] of complex

is one possible way to declare an array containing 400 items of type complex. A reference to any item requires 2 indexes. We can calculate the address of any item in a constant time. So reading or modifying a value can be taken to be an elementary operation. In the case of an $n \times n$ matrix, operations such as initializing every item or finding the largest item takes a time in $\theta(n^2)$.

5.2 Records and Pointers:

While an array has a fixed number of items of the same type a Record is a data structure comprising a fixed number of items, often called "fields" in this context, that are possibly of different types. For example, info about a person that interest us is name, age, weight and sex, we might want to use a data structure of the following form.

```
type person = record
        name : string
        age : integer
        weight : real
        male : Boolean
```

If "Steve" is a variable of this type, we use the dot notation to refer to the fields; for example, "Steve.name" is a string and "Steve.weight" is a real.

An array can appear as an item of a record, and records can be kept in arrays. If we declare, for example

class : array [1..50] of person

then the array class holds 50 records. The attributes of the seventh member of the class are referred to as class[7].name, class[7].age, and so on. As with arrays, provided a record holds only fixed-length items, the address of any particular item can be calculated in constant time, so consulting or modifying the value of a field may be considered as an elementary operation.

Many programming languages allow records to be created and destroyed dynamically. This is one reason why records are commonly used in conjunction with pointers. A declaration such as

type boss = ↑person

says that boss is a pointer to a record whose type is person. Such a record can be created dynamically by a statement such as

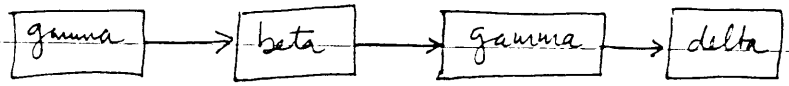
boss ← new person.

Now boss↑ means "the record that boss points to". To refer to the fields of this record, we use boss↑.name, boss↑.age and so on. If a pointer has the special value nil, then it is not currently pointing to any record.

5.3 LISTS:

A list is a collection of items of information arranged in a certain order. Unlike arrays and records, the number of items in a list is generally not fixed, nor is it bounded in advance. The corresponding data structure should allow us to determine, for example, which is the first item in the structure which is the last, which are the predecessor which are the successor (if they exist) of any given item. The storage corresponding to any given item of information is often called a node. Besides the info in question, a node may contain one or more pointers. The following is an example of a graphical

representation of a list:



Lists are subject to a number of operations: we might want to insert an additional node, to copy a list, to count the number of elements it contains, and so on. The various computer implementations commonly used differ in the quantity of storage ~~used~~ required, and in the ease of carrying out certain operations.

Implemented as an array by the declaration

```
type list = record
        counter : 0 .. maxlength
        value : array [1 .. maxlength] of information
```

the items of a list occupy the slots value[1] to value[counter], and the order of the items is the same as the order of the indexes in the array. Using this implementation, we can find the first and the last items of the list rapidly, as we can find the predecessor and the successor of a given item. As we mentioned before, inserting a new item or deleting one requires a worst-case number of operations in the order of the current size of the list. It was noted that this implementation is particularly efficient for the important structure known as stac and a stack can be considered as a kind of a list where addition and deletion of items are allowed only at one designated end of the list. Despite this, such an implementation of a stack may present the major disadvantage of requiring that all the storage potentially required be reserved throughout the life of the program.

On the other hand, if pointers are used to implement a list

structure, the nodes are usually records with a form similar to the following:

```
type list = ↑node
type node = record
              value : information
              next : ↑node
```

where each node except the last includes an explicit pointer to its successor. The pointer ~~is~~ the last node has the special value nil, indicating that it goes nowhere. In this case, storage needed to represent the list can be allocated and recovered dynamically. Hence, two or more lists may share the same list space, furthermore there is no need to know a priori how long any a particular list will get.

Even if additional pointers are used to ensure rapid access to the first and last items of the list, it is difficult when this representation is used to examine the k-th item, for arbitrary k, without having to follow k pointers and thus to take a time in $O(k)$. However, once an item has been found, inserting or deleting a node can be done rapidly by copying or changing just a few fields.

5.4 GRAPHS:

$G = (N, A)$ N: set of Nodes, A: set of arcs or edges

Concepts to be reviewed: Paths - Cycles - Connectedness - Strongly connected - Simple Graph.

There are at least 2 obvious ways to represent a graph on a computer. The first uses the concept of an adjacency matrix:

```
type adjgraph = record
                  value : array [1..nbnodes] of info
                  adjacent : array [1..nbnodes, 1..nbnodes] of Bool
```

If there is an edge from node i to j, then adjacent[i,j] = true otherwise false. If the graph is undirected, the matrix is necessarily symmetric.

With this representation it is easy to see whether or not there is an edge between 2 nodes. To look up a value in an array takes constant time. On the other hand, should we wish to examine all the nodes connected to some given node, we have to scan a complete row of the matrix, in the case of an undirected graph, or both a complete row and a complete column in the case of a digraph. This takes a time in $\theta(nbnodes)$, the number of nodes in the graph, independent of the number of edges, that enter or leave this particular node. The space required to represent a graph in this fashion is quadratic in the number of nodes.

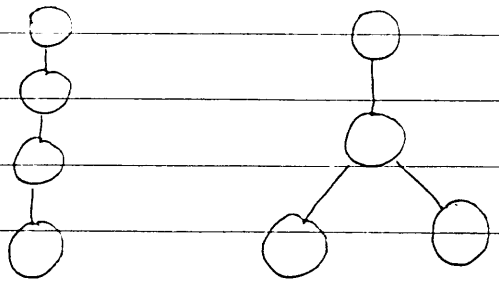
A second possible representation is the following.

```
type listgraph = array [1..nbnodes] of record
                    value : information
                    neighbors : list .
```

Here we attach to each node i a list of its neighbors, that is, of those nodes j such that an edge from i to j (for a digraph), or between i and j (for an undirected graph), exists. If the number of edges in the graph is small, this representation uses less storage than the one given previously. It is possible in this case to examine all the neighbors of a given node in less than nbnodes operations on the average. On the other hand, determining whether a direct connection exists between two given nodes i and j requires us to scan the list of neighbors of node i (and possibly of node j too, in the case of a digraph) which is less efficient than looking up a boolean value in an array.

5.5 TREES :

A tree is an acyclic connected undirected graph. There exists exactly one path between any 2 nodes. Example:

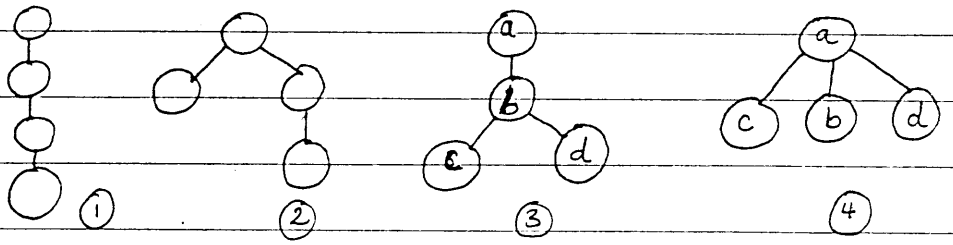


These are the only distinct possible trees we can have with 4 nodes

Trees have a number of simple properties of which the following are perhaps the most important:

- A tree with n nodes has exactly $n-1$ edges
- If an edge is added to a tree, then the resulting graph contains exactly one cycle.
- If a single edge is removed from a tree, then the resulting graph is no longer connected.

In our studies we'll be focusing on one type of tree, Rooted trees. These are special type of trees in which one node called "root" is special. When drawing the root is always on top, like a family tree. The following is an example of rooted trees (all possible rooted trees with 4 nodes).

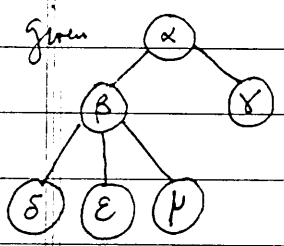


To describe the relationship between adjacent nodes, we use the concept of parent and child. in (3) a is the parent of b, c and d are children of b. Ancestors of d are a, b and d (this is not a typo, a node is always an ancestor of itself besides parent and parent's parent.) A descendent is defined analogously (again including the node itself).

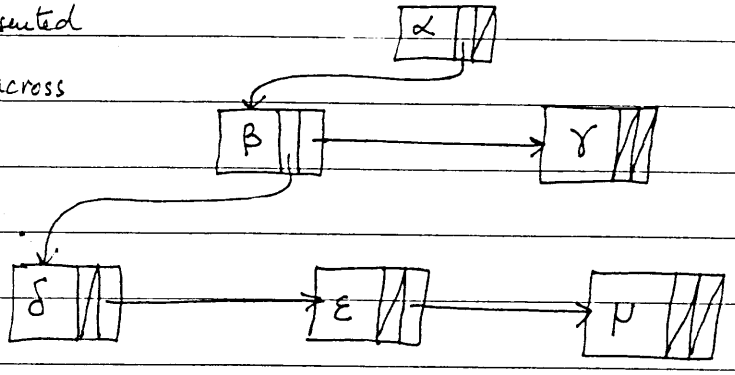
Again in (3) nodes c and d are leaves. b is an internal node
We consider branches of a rooted tree to be ordered from left to right: in (4) c is the eldest sibling of node b and d.
On a computer, any rooted tree may be represented using nodes of the following type.

```
type treenode1 = record
    value : information
    eldest-child, next sibling : ↑treenode1
```

Example:



this can be represented with structure across



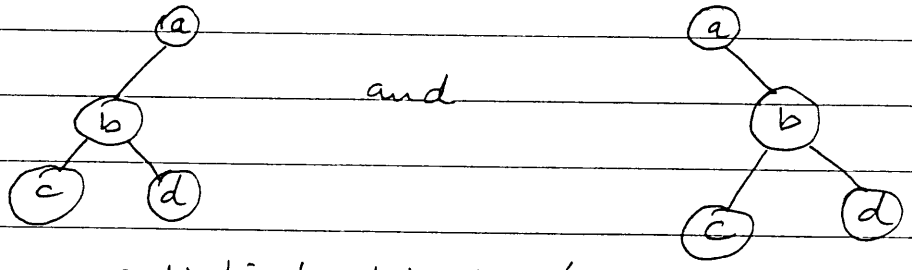
This representation can be used for any rooted tree; It has the advantage that all the nodes can be represented using the same record structure, no matter how many children or siblings they have. However many operations are inefficient using this minimal representation: it is not obvious how to find the parent of a given node.
Another representation suitable for any rooted tree uses nodes of the type:

```
type treenode2 = record
    value : information
    parent : ↑treenod2
```

where now each node contains only a single pointer leading to its parent. This representation is about as economical with storage space as one can hope for, but it is inefficient unless all the operations on the tree involve starting from a node and going up, never down.
A suitable representation for a particular application can

usually be designed by starting from one of these general representations and adding supplementary pointers, for example to the parent or to the eldest sibling of a given node. In this way we can speed up the operations we want to perform efficiently at the price of an increase in the storage needed.

We shall often have occasion to use binary trees. Each node has 0, 1, or 2 children. We assume a node has 2 pointers one to its left and one to its right, either of which can be nil. When we do this we talk about left child and right child and the position occupied by a child is significant. In the following example the 2 trees are considered different:



– 2 distinct binary trees –

We may generalize a Tree to a k-ary tree, where each node can have no more than k children and the representation uses nodes of the following type –

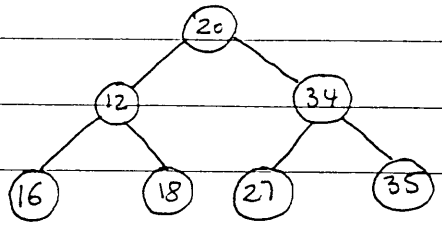
```
type k-ary-node = record
        value : information
        child : array [1..k] of ↑k-ary-node
```

In the case of a binary tree we can also define

```
type binary-node = record
        value : information
        left-child, right-child : ↑binary-node
```

A binary tree is a search tree if the value contained in every internal node is greater or equal to the values contained in its left child or any of that child's descendants, and less than or equal to the values contained in its right child or any of that child's descendants.

Example:



This structure is interesting because it allows efficient searches. In this example, although the tree has 7 items, we can find 27 with only 3 comparisons. The first, with value 20 at the root, tells us 27 is in the right subtree (if it is anywhere); the second 34 tells us to look down to the left; and the third finds the value we seek. This search procedure can be described more formally as follows.

```
function search (x, r)
{r is a pointer that points to the root. The function searches for x
 in this tree. If x is missing, the function returns nil.}
  if r = nil then return nil
  else if x = r↑.value then return r
  else if x < r↑.value then return search(x, r↑.left-child)
  else return search (x, r↑.right-child)
```

It is simple to update a search tree, that is, to delete a node or to add a new value without destroying the search tree property. However, if this is done carelessly, the resulting tree can become unbalanced. By this it is meant that many of the nodes in the tree have one child, not two, so its branches become long and stringy

When this happens, searching the tree is no longer efficient. In the worst case, every node in the tree may have exactly one child, except for a single leaf.

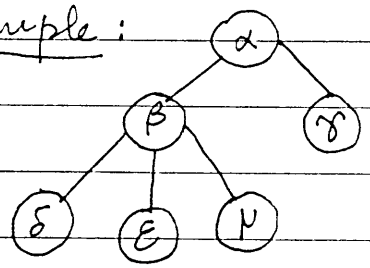
A variety of methods are available to keep the tree balanced, and hence to guarantee that such operations as searches or the addition and deletion of nodes take a time in $O(\log n)$ in the worst case, where n is the number of nodes in the tree.

These methods allow the efficient implementation of several additional operations. Among the older techniques are the use of AVL trees, and 2-3 trees; more recent techniques include: Red-Black trees and Splay trees.

HEIGHT, DEPTH and LEVEL

* The height of a node is the number of edges in the longest path from the node in question to a leaf.
* The depth of a node is the number of edges in the path from the root to the node in question
* The level of a node is equal to the height of the root of the tree minus the depth of the node concerned.

Example:



Node	Height	Depth	Level
α	2	0	2
β	1	1	1
γ	0	1	1
δ	0	2	0
ε	0	2	0
ν	0	2	0

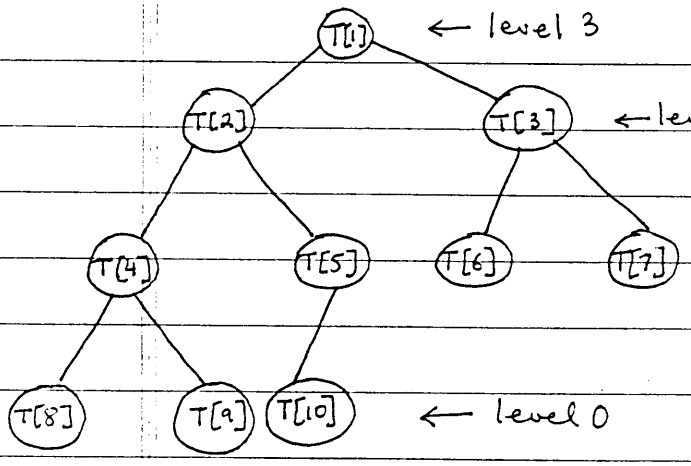
Informally, if the tree is drawn with successive generations of nodes in neat layers, then the depth of a node is found by numbering the layers downwards from 0 at the root; the level of a node is found by numbering the layers upward from 0 at the bottom; only the height is a little more complicated.

5.6 HEAPS:

A heap is a special kind of rooted tree that can be implemented efficiently in an array without any explicit pointers. It has numerous applications especially a sorting technique called heapsort. It is also used for the efficient representation of certain dynamic priority lists, such as the list of tasks to be scheduled by an operating system.

. A binary tree is essentially complete if each internal node, with possible exception of one special node, has 2 children. The special node if it exists is situated on level 1. It has a left child but no right child. Moreover, either all the leaves are on level 0, or else they are on levels 0 and 1, and no leaf on level 1 is to the left of an internal node at the same level.

Example: an essentially complete binary tree with 10 nodes



The 5 internal nodes occupy level 3, 2 and 1. The 5 leaves ~~occupy~~ fill the right side of level 1 and level 0.

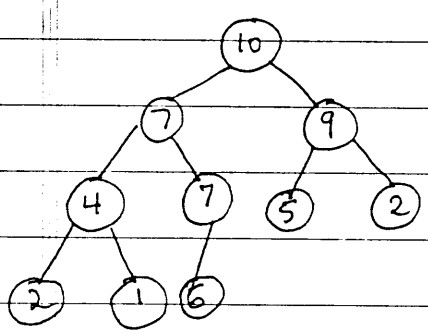
If an essentially complete binary tree (ECBT) has height k, then there is one node (the root) on level k, there are 2 nodes on level $k-1$, and so on; there are 2^{k-1} nodes on level 1, and at least 1 and not more than 2^k on level 0. If the tree contains n nodes in all, it follows that $2^k \leq n \leq 2^{k+1}$. Equivalently, the height of a tree containing n nodes is $k = \lfloor \log n \rfloor$, a result we will use later.

This kind of tree can be represented in an array T by putting the nodes of depth k, from left to right, in the positions $T[2^k], T[2^k+1], \ldots, T[2^{k+1}-1]$ with the possible exception of level 0, which may be incomplete.

The parent of the node represented in $T[i]$ is found in $T[i \div 2]$ for $i > 1$ (the root $T[1]$ does not have a parent), and the children of the node represented in $T[i]$ are found in $T[2i]$ and $T[2i+1]$ whenever they exist of course.

Now a heap is an essentially complete binary tree (ECBT), each of whose nodes includes an element of information called the value of the node, and which has the property that the value of each internal node is greater than or equal to the values of its children. This is called the heap property.

Example: A heap with 10 nodes.



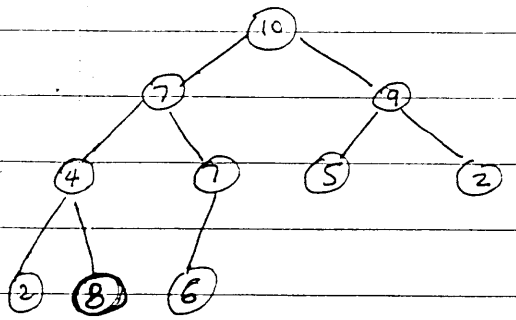
This heap can be represented also by the following array

10 7 9 4 7 5 2 2 1 6

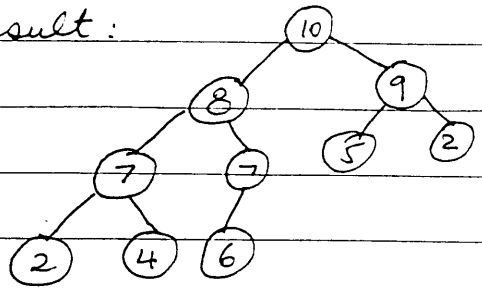
The crucial characteristic of this data structure is that it can be restored efficiently if the value of a node is modified. If the value of a node increases till it becomes greater that its parent, it suffices to exchange the values, and continue the process upwards in the tree if necessary until the heap property is restored. We say that the modified value has been percolated up to its new position in the heap. The operation itself is called sifting up.

Example:

I substitute 1 with 8 in the previous graph.

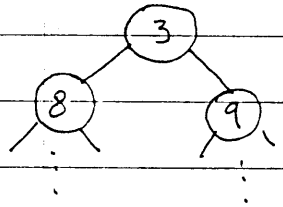


restore the heap property by exchanging 8 with its parent 4, and then exchanging it again with its new parent 7, obtaining the result:



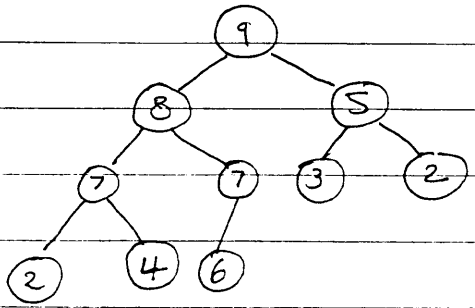
If the value of a node is decreased, so that it becomes less than the value of at least one of the children, it suffices to exchange the modified value with the larger of the values in the children, and then continue this process downwards in the tree if necessary till the heap property is restored. We say that the modified value has been sifted down to its new position.

Example: Take the previous example and modify the root 1 with 3. The top part of the tree looks like:



To restore the heap property, exchange 3 with the larger child, 9. Then again with the larger of the new children, 5.

and finally we obtain:



The following procedures describe the basic processes for manipulating a heap.

```
procedure alter-heap (T[1..n], i, v)
    {T[1..] is a heap. T[i] is set to v and the heap property
    is re-established. We suppose that 1 ≤ i ≤ n.}
    x ← T[i]
    T[i] ← v
    if v < x then sift-down (T, i)
             else percolate (T, i)

procedure sift-down (T[1..n], i)
{This procedure sifts node i down so as to re-establish
the heap property in T[1..n].
We suppose that
1 ≤ i ≤ n.}
k ← i
repeat
    j ← k      {find the larger child of j}
    if 2j ≤ n and T[2j] > T[k] then k ← 2j
    if 2j < n and T[2j+1] > T[k] then k ← 2j+1
    exchange T[j] and T[k]
    {if j = k, then the node has arrived at its final position}
until j = k

procedure percolate (T[1..n], i)
    k ← i
    repeat
        j ← k
        if j > 1 and T[j÷2] < T[k] then k ← j÷2
        exchange T[j] and T[k]
    until j = k
```

The heap is an ideal data structure for finding the largest element of a set, removing it, adding a new node, or modifying a node. These are exactly the operations we need to implement dynamic priority lists efficiently: the value of a node gives the priority of the corresponding event, the event with highest priority is always found at the root of the heap, and the priority of an event can be changed dynamically at any time. This is particularly useful in computer simulations and in the design of schedulers for an operating system.

Some typical procedures:

(1)
```
. function find-max (T[1..n])
      return T[1]      { returns the largest element }
```

(2)
```
. procedure delete-max (T[1..n])
      T[1] ← T[n]                    { remove the largest and
      sift-down (T[1..n-1], 1)         restore the heap property }
```

(3)
```
. procedure insert-node (T[1..n], v)
      T[n+1] ← v
      percolate (T[1..n+1], n+1)
```

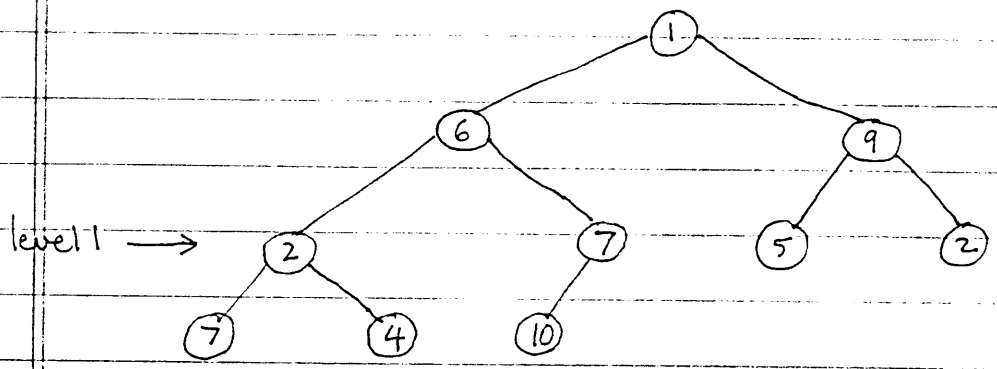
What we need to know now, is how to create a heap starting from an array T[1..n] of elements in an undefined order. The obvious solution is to start with an empty heap and to add elements one by one as shown in the following

```
. procedure slow-make-heap (T[1..n])
      for i ← 2 to n do {percolate} (T[1..i], i)
```

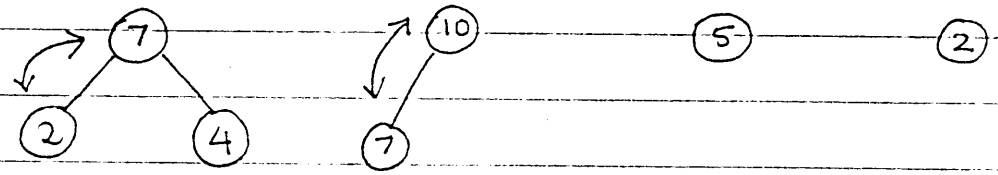
However, this approach is inefficient. There is a better way for making a heap. Suppose we start with the following array:

1 6 9 2 7 5 2 7 4 10

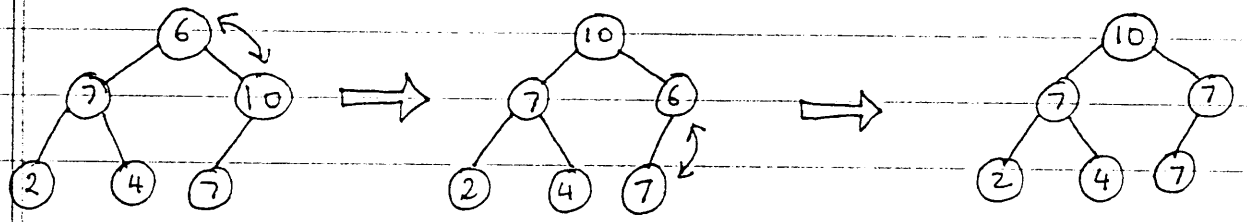
represented by the following tree.



We first make each of the subtrees whose roots are at level 1 into a heap, this is done by sifting down these roots as shown below:



Again at the next higher level (shown below for the left subtree)



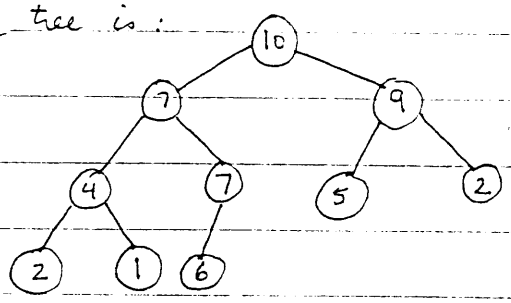
The other subtree at level 2, is already a heap. This result is an essentially complete binary tree corresponding to the array

1 10 9 7 7 5 2 2 4 6

It only remains to sift down its root to obtain the desired heap. The final process goes as follows:

10	1	9	7	7	5	2	2	4	6
10	7	9	1	7	5	2	2	4	6
10	7	9	4	7	5	2	2	1	6

and the corresponding tree is:



Here is the formal description of the algorithm

```
procedure make-heap (T[1..n])
    {This procedure transforms T[1..n] into a heap}
    for i ← ⌊n/2⌋ downto 1 do sift-down (T, i)
```

The algorithm constructs a heap in linear time.

Williams invented the heap to serve as the underlying data structure for the following sorting algorithm:

```
procedure heapsort (T[1..n])
    {T is an array to be sorted}
    make-heap (T)
    for i ← n downto 2 do
        exchange T[1] and T[i]
        siftdown (T[1..i-1], 1)
```

The algorithm takes a time in $O(n \log n)$ to sort n elements.

$t(n)$ = make heap takes linear time n + $(n-1)$ exchanges + $(n-1)$ siftdown operations (sifts down a path whose length is at most $\log n$), thus:

$$t(n) \in O(n) + (n-1)\,O(1) + (n-1)\,O(\log n) = O(n \log n)$$