Recursive Algorithms

- In this technique, we define a procedure that is allowed to make calls to itself as a subroutine
- Those calls are meant to solve sub-problems of smaller size
- Recursive procedure should always define a base case that can be solved directly without using recursion

Recursive procedure

Algorithm recursiveMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

if $n = 1$ **then**

return $A[0]$

return $\max\{\text{recursiveMax}(A, n-1), A[n-1]\}$

Recurrence Equation

- Recurrence equation defines mathematical statements that the running time of a recursive algorithm must satisfy
- Function *T(n)* denotes the running time of the algorithm on an input size n, e.g.,

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ T(n-1) + 7 & \text{otherwise,} \end{cases}$$

- Ideally, we would like to characterize a recurrence equation in closed form, e.g., $T(n)=7(n-1)+3 =7n -2 = O(n)$

Data Structures

- An important element in the design of any algorithmic solution is the *right choice of the data structure*
- Data structures provide some mechanism for *representing sets and operations* defined on *set elements*
- Some basic and general data structures appear as *elements of programming languages*, e.g., as types: arrays, strings, sets, records, …)
- Some other: *abstract data structures* are more specialised and complex (stacks, queues, lists, trees, graphs, …)

Data Structures

- Typical operations defined on data structures:
  - checking (set) membership
  - accessing indexed elements
  - insertion/deletion/update
  - more complex (set of objects) querying
- The *efficiency of operations* provided by the data structures is usually related to the *level of ordering* of stored data.

Stacks

- *Objects* can be *inserted* into a stack *at any time*, but only the *most recently inserted* ("last") object can be *removed at any time*
- E.g., Internet Web browsers store the address of recently visited sites on a stack
- A stack is a container of objects that are inserted according to the last in first out (LIFO) principle

Stack Abstract Data Type

- A *stack* is an abstract data type (ADT) supporting the following two methods
  - *push(o)* : insert object o at the top of the stack
  - *pop()* : remove from the stack and return the top object on the stack; an error occurs if the stack is empty

Stack (supporting methods)

- The stack supporting methods are:
  - *size()* : return the number of objects in the stack
  - *isEmpty()* : return a Boolean indicating if the stack is empty
  - *top()* : return the top object on the stack, without removing it; an errors occurs if the stack is empty

Stack (array implementation)

- A *stack* can be implemented with an N-element array *S*, with elements stored from *S[0]* to *S[t]*, where *t* is an integer that gives the index of the top element in S

Stack Main Methods

Algorithm push(o):
  if size() $= N$ **then**
    indicate that a stack-full error has occurred
  $t \leftarrow t + 1$
  $S[t] \leftarrow o$

Algorithm pop():
  if isEmpty() **then**
    indicate that a stack-empty error has occurred
  $e \leftarrow S[t]$.
  $S[t] \leftarrow$ **null**
  $t \leftarrow t - 1$
  return e

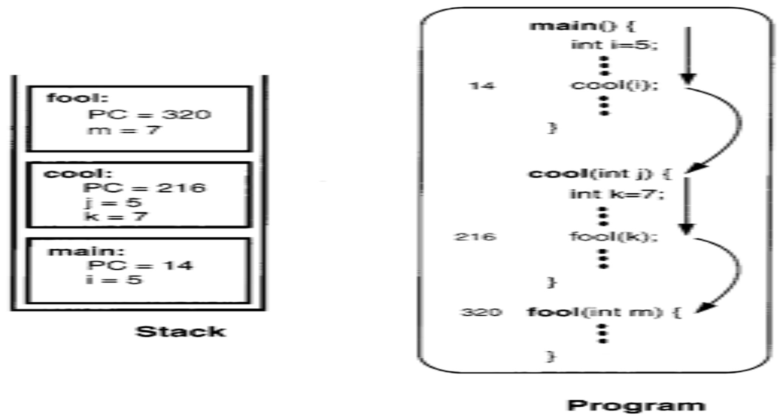
Stack Methods Complexity

- each of the *stack methods* executes a *constant* number of statements
- all *supporting methods* of the Stack ADT can be easily implemented in constant time
- thus, in array implementation of stack ADT *each method* runs in *O(1)* time

Stack (application)

- Stacks are important application to the run-time environments of modern procedural languages (C,C++,Java)
- Each thread in a running program written in one of these languages has a private stack, method stack, which is used to keep track of local variables and other important information on methods

Stack (application)



Stack (recursion)

- One of the benefits of using stack to implement method invocation is that it allows programs to use recursion
- *Recursion* is a powerful method, as it often allows to design *simple and efficient* programs for fairly *difficult problems*

Queues

- A queue is a container of objects that are inserted according to the first in first out (FIFO) principle
- *Objects* can be *inserted* into a queue *at any time*, but only the element that was in the queue the *longest* can be *removed at any time*
- We say that elements *enter* the queue at the rear and are *removed* from the front

Queue ADT

- The queue ADT supports the following two fundamental methods
  - *enqueue(o)* : insert object o at the rear of the queue
  - *dequeue(o)* : remove and return from the queue the object at the front; an error occurs if the queue is empty

Queue (supporting methods)

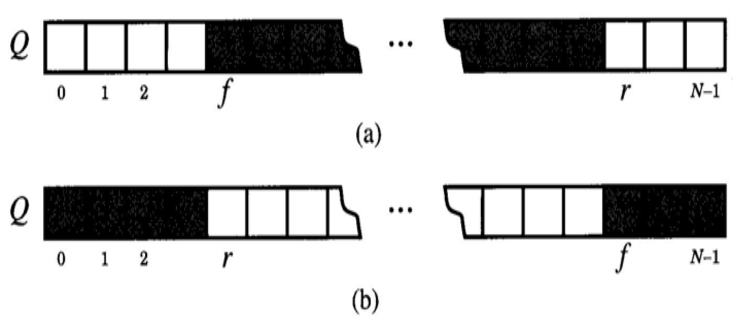
- The queue supporting methods are
  - *size()* : return the *number* of objects in the queue
  - *isEmpty()* : return a *Boolean* value indicating whether the queue is empty
  - *front()* : return, but *do not remove*, the front object in the queue; an error occurs if the queue is empty

Queue (array implementation)

- A *queue* can be implemented an *N*-element array *Q*, with elements stored from *S[f]* to *S[r] (mod N)*
- *f* is an index of *Q* storing the *first* element of the queue (if not empty)
- *r* is an index to the *next available* array cell in *Q* (if *Q* is not full)

Queue (array implementation)

- Normal ($f \leq r$) configuration (a) and wrap around ($f > r$) configuration (b)



Queue (main methods)

Algorithm dequeue():
    if isEmpty() **then**
        throw a QueueEmptyException
    $temp \leftarrow Q[f]$
    $Q[f] \leftarrow$ **null**
    $f \leftarrow (f+1) \bmod N$
    return *temp*

Algorithm enqueue(o):
    if size() $= N - 1$ **then**
        throw a QueueFullException
    $Q[r] \leftarrow o$
    $r \leftarrow (r+1) \bmod N$

Queue Methods Complexity

- each of the *queue methods* executes a *constant* number of statements
- all *supporting methods* of the queue ADT can be easily implemented in constant time
- thus, in array implementation of queue ADT *each method* runs in *O(1)* time

Queue and Multiprogramming

- Multiprogramming is a way of achieving a limited form of *parallelism*
- It allows to run *multiple tasks* or computational threads *at the same time*
- E.g., one thread can be responsible for catching mouse clicks while others can be responsible for moving parts of animation around in a screen canvas

Queue and Multiprogramming

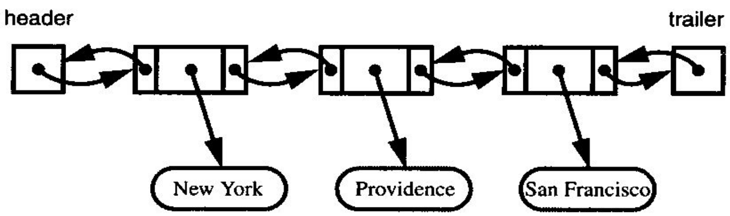
- When we design a *program* or operating system that uses *multiple threads*, we must disallow an individual thread to *monopolise* the CPU, in order to avoid *application* or *applet* hanging
- One of the solutions is to utilise a queue to allocate the CPU time to the running threats in the round-robin protocol.

Linked List

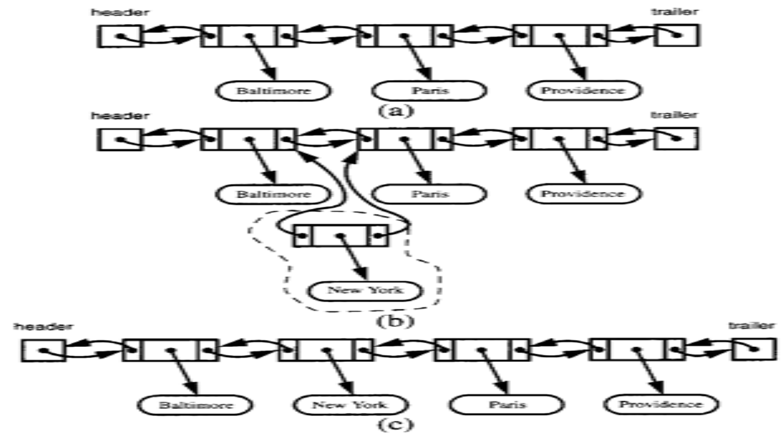
- A node in a singly linked list stores in a next link a reference to the *next node* in the list (traversing in only *one direction* is possible)
- A node in a doubly linked list stores two references – a next link, and a previous link which points to the *previous node* in the list (traversing in two *two directions* is possible)

Doubly Linked List

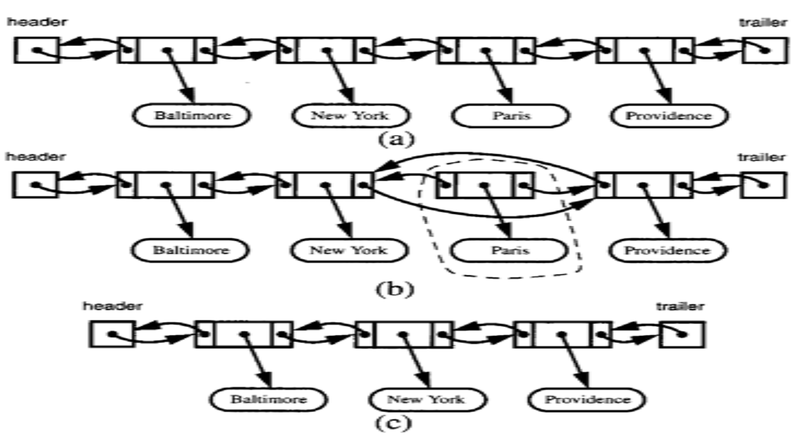
- *Doubly linked* list with two sentinel (dummy) nodes header and trailer



List Update (element insertion)



List Update (element removal)

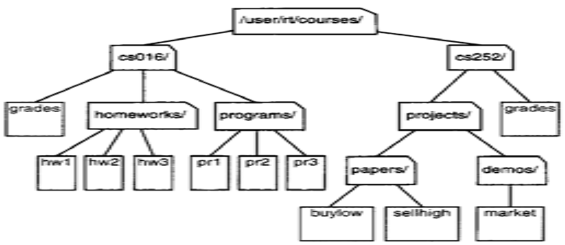


List Update (complexity)

- What is the cost (complexity) of both *insertion* and *removal* update?
  - If the *address* of element at *position p* is known, the cost of an update is *O(1)*
  - If *only* the *address* of a *header* is known, the cost of an update is *O(p)* (we need to traverse the list from position *0* up to *p*)

Rooted Tree

- A tree T is a set of nodes storing elements in a parent-child relationship, s.t.,
  - T has a special node r, called the root of T
  - Each node v of T different from r has a parent node u.



Rooted Tree

- If node u is a *parent* of node v, we say that v is a child of u
- Two nodes that are *children* of the *same parent* are siblings
- A node is external (leaf) if it has *no children*, and it is internal otherwise
- *Parent-child* relationship naturally extends to ancestor-descendent relationship
- A tree is ordered if there a *linear ordering* defined for the *children* of each node

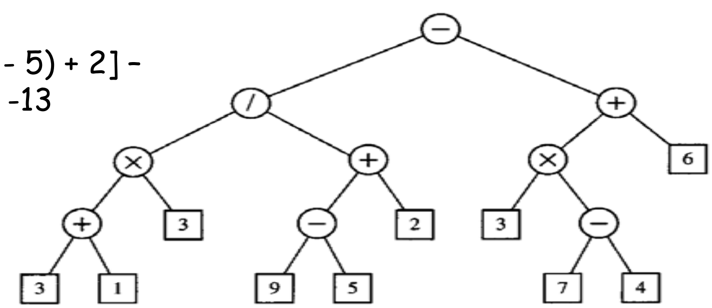
Binary Tree

- A binary tree is an ordered tree in which every node has *at most two children*
- A *binary tree* is proper if each internal node has *exactly two children*
- *Each child* in a binary tree is labelled as either a left child or a right child

Binary Tree (arithm. expression)

- *External node* is a *variable* or a *constant*
- *Internal node* defines *arithmetic operation* on its children

$[(3 + 1) \cdot 3]/[(9 - 5) + 2] - [3 \cdot (7-4) + 6] = -13$



The Depth in a Tree

- The depth of v is the number of ancestors of v, excluding v itself

```
Algorithm depth(T, v):
   if T.isRoot(v) then
      return 0
   else
      return 1 + depth(T, T.parent(v))
```


The Height of a Tree

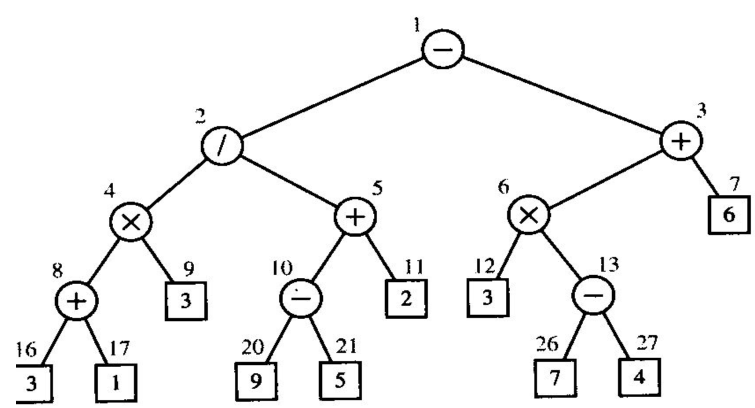
- The height of a tree is equal to the maximum depth of an external node in it

```
Algorithm height(T, v):
   if T.isExternal(v) then
      return 0
   else
      h = 0
      for each w ∈ T.children(v) do
         h = max(h, height(T, w))
      return 1 + h
```


Data Structures for Trees

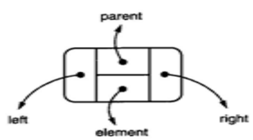
- Vector-based structure:
  - *v* is *the root -> p(v)* = 1
  - *v* is the *left child* of *u -> p(v)* = 2· *p(u)*
  - *v* is the *right child* of *u -> p(v)* = 2· *p(u) +1*
- The numbering function *p()* is known as a level numbering of the nodes in a binary tree.
- Efficient representation for *proper* binary trees

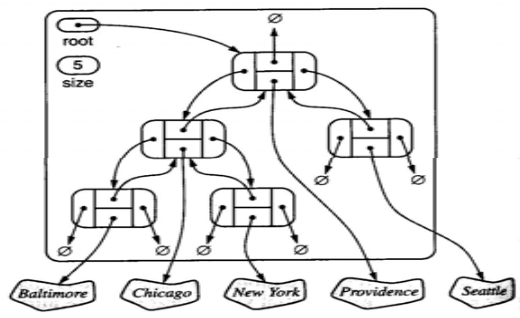
Data Structures for Trees



Data Structures for Trees

- Linked structure : each node v of T is represented by an *object* with *references* to the *element stored at v* and positions of its *parent* and *children*





Priority Queue

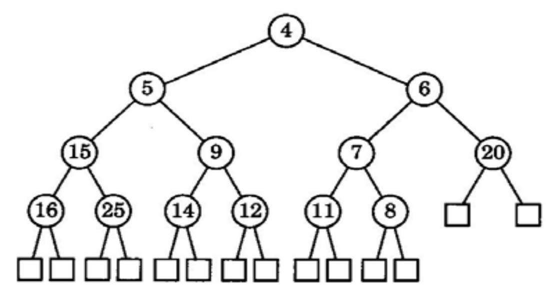
- Priority queue is an abstract data structure used to store elements from the ordered (≤) set
- The operations defined on priority queue *PQ*
  - *Create(PQ)* – creates empty priority queue *PQ*
  - *Insert(PQ, el)* – inserts element *el* to *PQ*
  - *RemoveMin(PQ)* – removes minimal element from *PQ*
  - *Min(PQ)* – gives the value of the minimal element

Heap Data Structure

- A heap is a realisation of PQ that is *efficient* for both *insertions* and *removals*
- heap allows to perform both *insertions* and *removals* in *logarithmic time*
- In heap the elements and their keys are stored in (almost *complete*) *binary tree*

Heap-Order Property

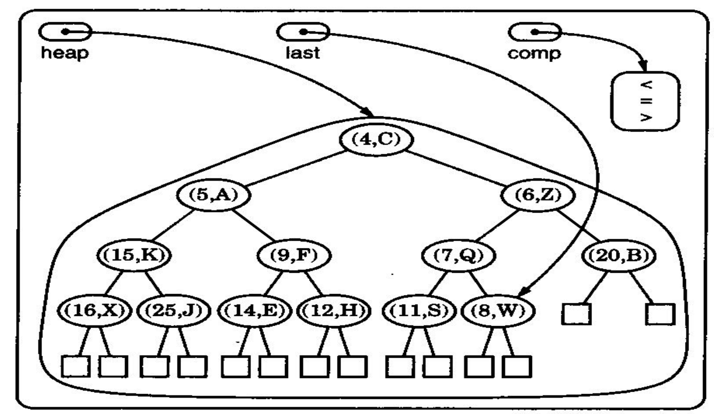
- In a heap *T*, for *every node v* other than the *root*, *the key* stored at *v* is greater than (or equal) to *the key* stored at its parent



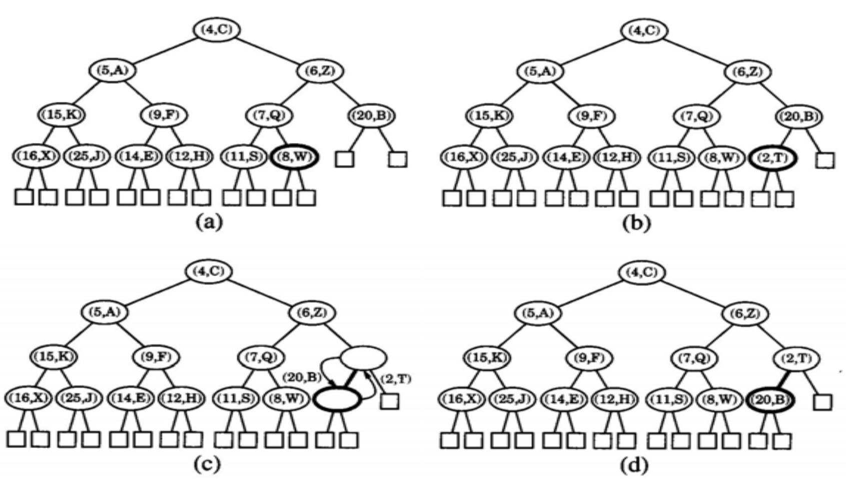
PQ/Heap Implementation

- heap: *complete binary tree T* containing *elements* with *keys* satisfying *heap-order property*; implemented using a *vector representation*
- last: *reference* to the *last used* node of *T*
- comp: *comparator* that defines the *total order relation* on *keys* and *maintains* the *minimum* element at the *root* of *T*

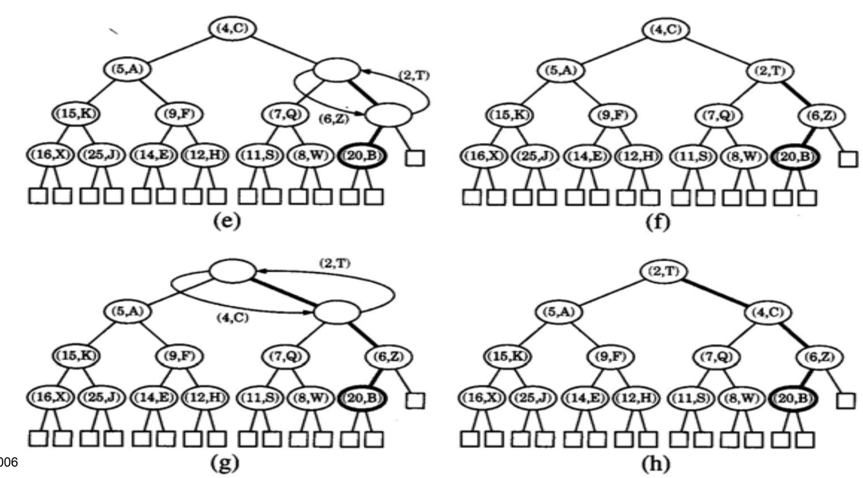
PQ/Heap Implementation



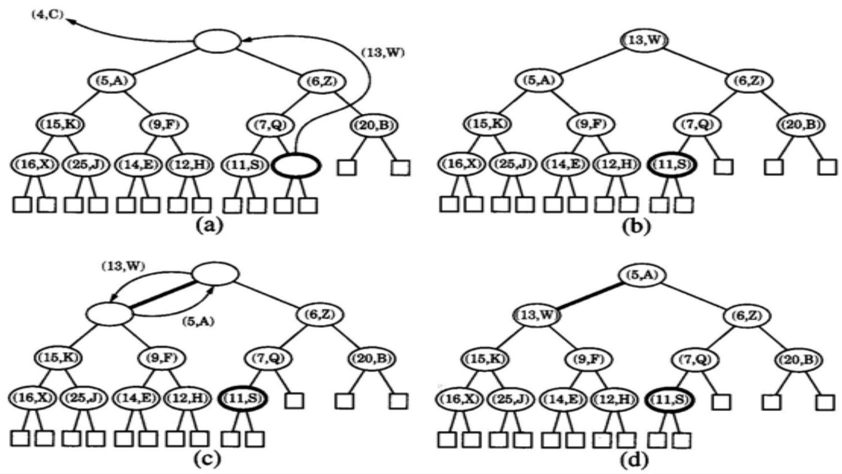
Up-Heap Bubbling (insertion)



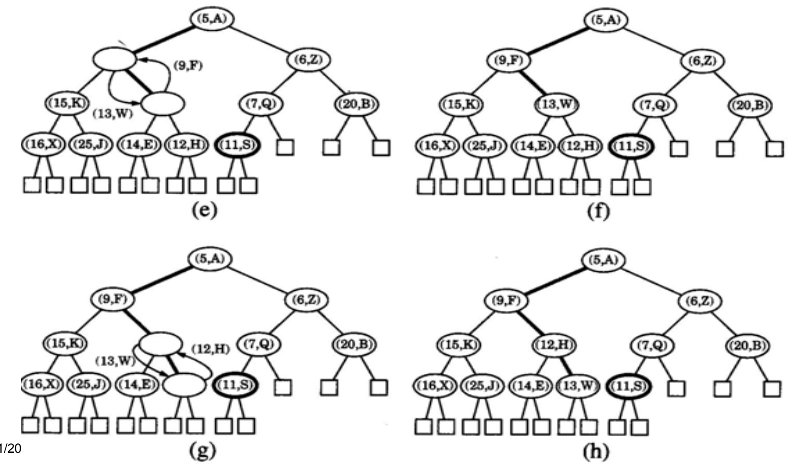
Up-Heap Bubbling (insertion)



Down-Heap Bubbling (removal)



Down-Heap Bubbling (removal)



Heap Performance

- Operation times:
  - *Create(PQ):* *O(1)*
  - *Min(PQ):* *O(1)*
  - *Insert(PQ, el):* *O(log n)*
  - *RemoveMin(PQ):* *O(log n)*
- Heaps have several applications including sorting (Heap-sort) and data compression (Huffman coding).

Heap-Sorting

- **Theorem:** The heap-sort algorithm sorts a sequence of *S* of *n* comparable elements, e.g., numbers, in time *O(n* log *n)*, where
  - Bottom-up construction of heap with *n* items takes *O(n)* units of time, and
  - Extraction of *n* elements (in increasing order) from the heap takes *O(n* log *n)* units of time

Representation of sets

- We already know that sets can be represented in many ways as different types of data structures
- Efficiency of set representation depends on its size and application
- Small sets can be represented as characteristic vectors (binary arrays), where:
  - the array is indexed by the set elements
  - the entries are either 1 (element is in) or 0 (otherwise)

Example

- A subset of the universal set U={0,1,2,3,4,5,6,7,8,9} can be represented as any binary array of length 10
- For example, the subset S of odd numbers from U, i.e., S={1,3,5,7,9} can be represented as:

0	1	0	1	0	1	0	1	0	1
0	1	2	3	4	5	6	7	8	9

Generation of all k-subsets

- Generation of all *k-subsets* of the universal set U={0,1,2,3,4,5,6,7,8,n-1} can be done with a help of the following formula (details to be discussed):

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Generation of all 3-subsets

1110000000	1001000001	0110000100	0100001001	0010001010	0000111000
1101000000	1000110000	0110000010	0100000110	0010001001	0000110100
1100100000	1000101000	0110000001	0100000101	0010000110	0000110010
1100010000	1000100100	0101100000	0100000011	0010000101	0000110001
1100001000	1000100010	0101010000	0011100000	0010000011	0000101100
1100000100	1000100001	0101001000	0011010000	0001110000	0000101010
1100000010	1000011000	0101000100	0011001000	0001101000	0000101001
1100000001	1000010100	0101000010	0011000100	0001100100	0000100110
1011000000	1000010010	0101000001	0011000010	0001100010	0000100101
1010100000	1000010001	0100110000	0011000001	0001100001	0000100011
1010010000	1000001100	0100101000	0010110000	0001011000	0000011100
1010001000	1000001010	0100100100	0010101000	0001010100	0000011010
1010000100	1000001001	0100100010	0010100100	0001010010	0000011001
1010000010	1000000110	0100100001	0010100010	0001010001	0000010110
1010000001	1000000101	0100011000	0010100001	0001001100	0000010101
1001100000	1000000011	0100010100	0010011000	0001001010	0000010011
1001010000	0111000000	0100010010	0010010100	0001001001	0000001110
1001001000	0110100000	0100010001	0010010010	0001000110	0000001101
1001000100	0110010000	0100001100	0010010001	0001000101	0000001011
1001000010	0110001000	0100001010	0010001100	0001000011	0000000111