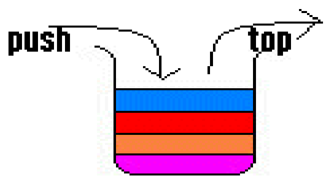
# Elementary Data Structures

## Stacks, Queues, Lists, and Related Structures

**Stacks, lists and queues are primitive data structures fundamental to implementing any program requiring data storage and retrieval. The following tables offer specific information on each type of data structure. The rest of the web page offers information about implementing and applying these data structures.**

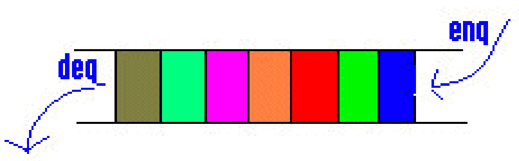
| ADT | Mathematical Model | Operations |
|---|---|---|
| Stack |  | Push(S,Data)<br>Pop(S)<br>Makenull(S)<br>Empty(S)<br>Top(S) |

The mathematical model of a stack is LIFO (last in, first out). Data placed in the stack is accessed through one path. The next available data is the last data to be placed in the stack. In other words, the "newest" data is withdrawn.

The standard operations on a stack are as follows:

| | |
|---|---|
| PUSH(S, Data) : | Put 'Data' in stack 'S' |
| POP(S) : | Withdraw next available data from stack 'S' |
| MAKENULL(S) : | Clear stack 'S' of all data |
| EMPTY(S) : | Returns boolean value 'True' if stack 'S' is empty; returns 'False' otherwise |
| TOP(S) : | Views the next available data on stack 'S'. This operation is redundant since one can simply POP(S), view the data, then PUSH(S,Data) |

All operations can be implemented in O(1) time.

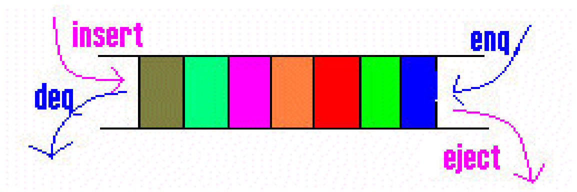
| ADT | Mathematical Model | Operations |
|---|---|---|
| Queue |  | Enqueue(Q)<br>Dequeue(Q)<br>Front(Q)<br>Rear(Q)<br>Makenull(Q) |

**The mathematical model of a queue is FIFO(first in, first put). Data placed in the queue goes through one path, while data withdrawn goes through another path at the "opposite" end of the queue. The next available data is the first data placed in the queue. In other words, the "oldest" data is withdrawn.**

**The standard operation on a queue are as follows:**

| | |
|---|---|
| **ENQUEUE(Q, Data) :** | **Put 'Data' in the rear path of queue 'Q'** |
| **DEQUEUE(Q) :** | **Withdraw next available data from front path of queue 'Q'** |
| **FRONT(Q) :** | **Views the next available data on queue 'Q'** |
| **REAR(Q) :** | **Views the last data entered on queue 'Q'** |
| **MAKENULL(Q) :** | **Clear queue 'Q' of all data.** |

**All operations can be implemented in O(1) time.**

| ADT | Mathematical Model | Operations |
|---|---|---|
| Deque |  | Inject(D,Data)<br>Eject(D)<br>Dequeue(D)<br>Enqueue(D,Data)<br>Front(D)<br>Rear(D)<br>MakeNull(D) |

The mathematical model of a deque is similar to a queue. However, a deque is a "double-ended" queue. The model allows data to be entered and withdrawn from the front and rear of the data structure.

The standard operations on a deque are as follows:

| | |
|---|---|
| INJECT(D, Data) : | Put 'Data' in front path of deque 'D' |
| EJECT(D) : | Withdraw next available data from rear path of deque 'D' |
| ENQUEUE(D, Data) : | Put 'Data' in rear path of deque 'D' |
| DEQUEUE(D) : | Withdraw next available data from front path of deque 'D' |
| FRONT(D) : | Views the next available data from front path of deque 'D' |
| REAR(D) : | Views the next available data from rear path of deque 'D' |
| MAKENULL(D) : | Clear deque 'D' of all data. |

All operations can be implemented in O(1) time.

| ADT | Mathematical model | Operations |
|---|---|---|
| List | $[x_1, x_2, ..., x_n]$ | Concatenate(L1, L2)<br>Access(L, i)<br>Sublist(L, [i...j]) |

**The mathematical model of a list is a string of data. The model allows data to be added or deleted anywhere in the list.**

**The standard operations on s list are as follows:**

| | |
|---|---|
| **CONCATENATE(L1, L2) :** | **Two lists L1, L2 are joined as follows,**<br>**L1 = $[x_1, x_{2,} x_3, ..., x_n]$**<br>**L2= $[y_1, y_2, y_3,...,y_m]$**<br><br>**L = $[x_1, x_2, x_3,...,x_n, y_1, y_2, ...,y_m]$** |
| **ACCESS(L, i) :** | **Returns data $x_i$** |
| **SUBLIST(L, [i...j]) :** | **Returns $[x_i, x_{i+1}, ...,x_j]$ Variations of this are sublist(L,[i...])which returns $[x_i, x_{i+1}, ...x_n]$, and SUBLIST(L,[...i]), which returns $[x_1, x_2,...,x_i]$** |

**CONCATENATE, ACCESS, and SUBLIST are called "ATOMIC" operations. Using these three operations we can make other operations.**

**Examples:**

**INSERT_AFTER(L, x, i) : Inserts data 'x' after member $x_i$ in list 'L'.**
**This can be inplemented by the following operations:**
**CONCATENATE(SUBLIST(L,[...i])), CONCANTENATE([x], SUBLIST(l,[I+1...]))**

**DELETE(L, i) :Removes $x_i$ from list 'L'.**
**This can be implemented as the following operations:**
**CONCATENATE(SUBLIST(L[...i-1])), SUBLIST(L, [i+1...]))**

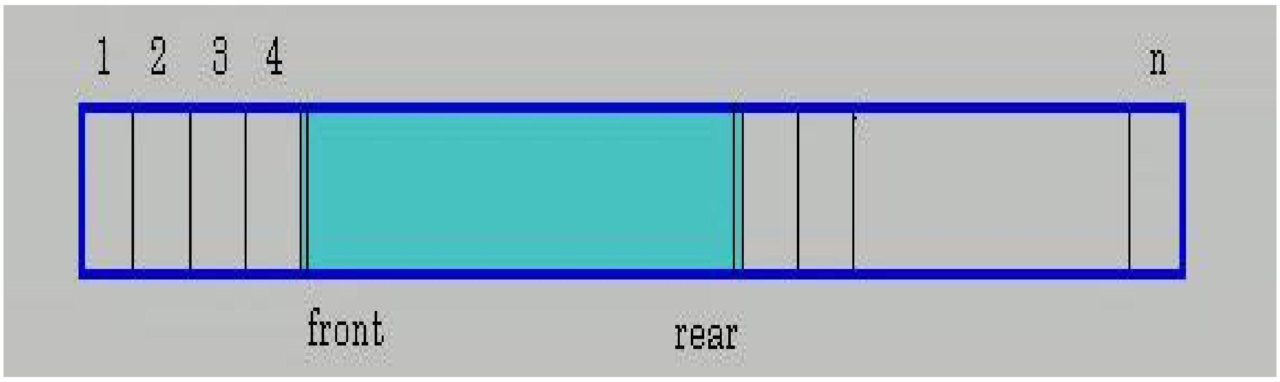
**The atomic operations may be used to describe the standard stack, queue and deque operations.**

## Implementations

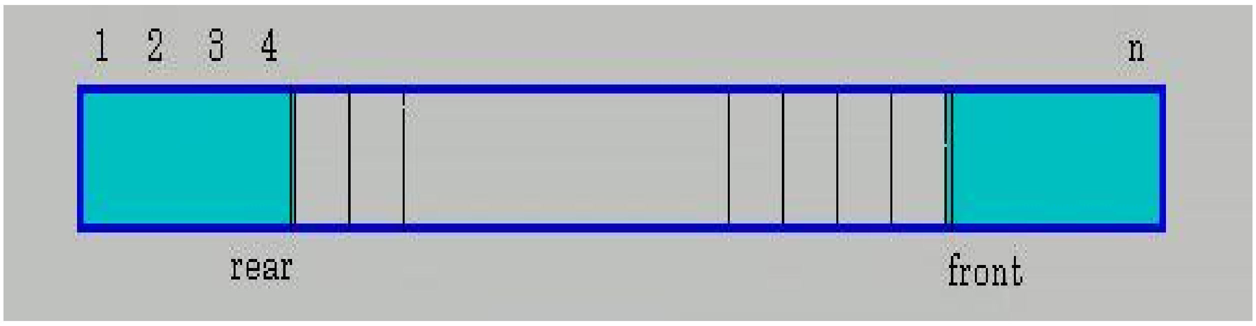
**The ADTs can be implemented by various data structures.**

**1. Tables/Arrays:**

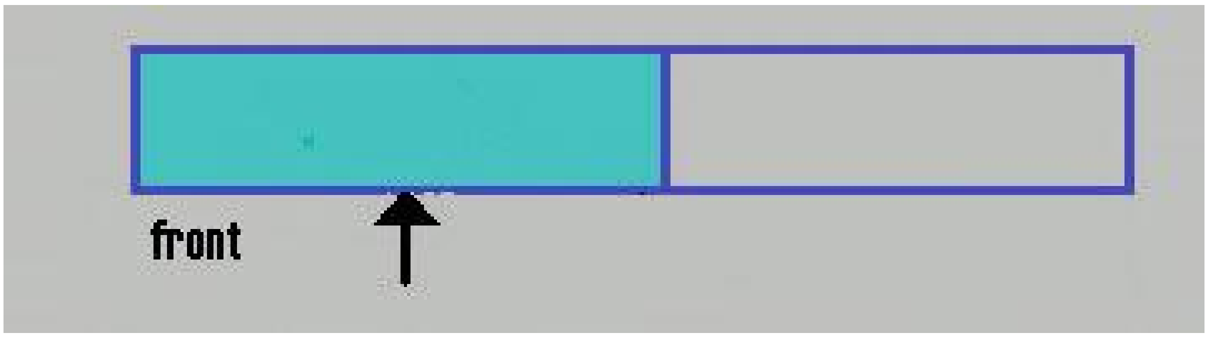
**A static array can be used to implement the data structures. Consider, for example, a queue implemented as an array.**



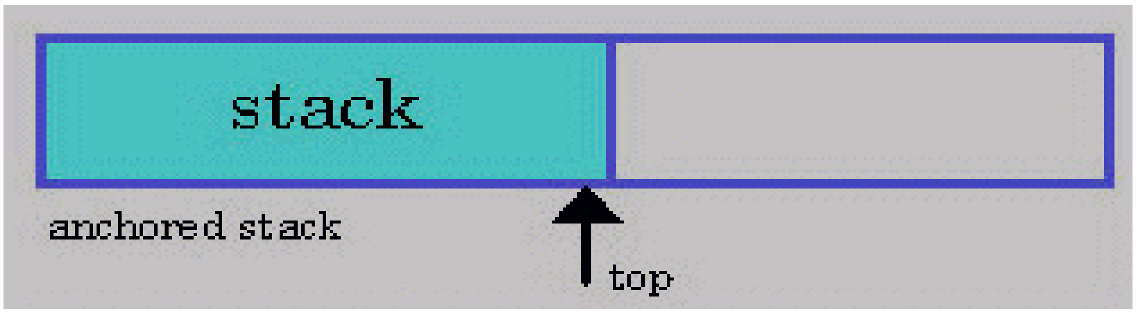
**As data is added to the rear of the queue, the cell labelled 'rear' (the cell containing the last enqueued data) is incremented to the right. If data is dequeued, the cell labelled 'front' (the cell containing the first enqueued data) is also incremented to the right. Subsequently, the data could end up flanked by empty cells.**



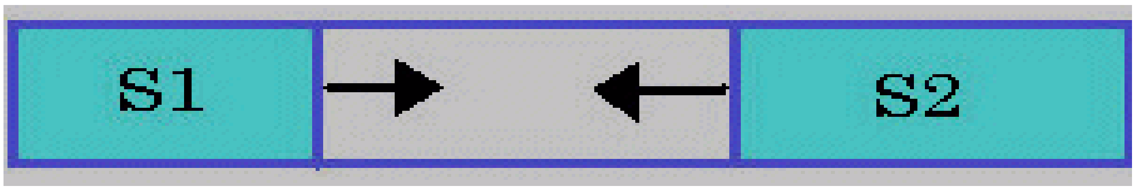
**If the cell labelled 'rear' is the last cell in the array, AND there are empty cells at the beginning of the array (due to previous dequeues) then 'rear' will wrap to the beginning of the array on the next enqueue. This will result in data at the beginning and end of the array, with empty cells in the middle.**



**An ANCHORED LIST prevents this movement of data in the array. Data is always left skewed. In our queue example, if data is dequeued then the whole array must be shifted one cell to the left.**



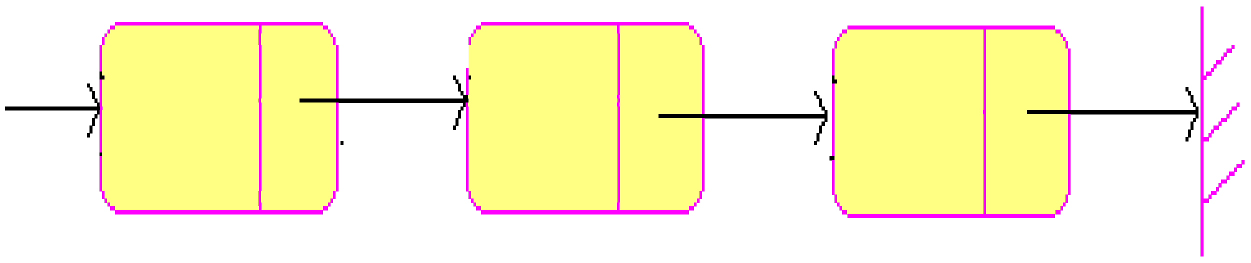
**Anchored lists are more appropiate for stacks, where the non-anchored end of the list represents the top of the stack.**



Two stacks can be anchored at opposite ends of an array. As the stacks fill with data, they will "grow" towards each other. The above figure illustrates this concept. By filling the array with the two stacks anchored at opposite ends, the user can have the utility of two stacks while using the storage of one array.

## 2. Linked Lists:

A series of structures connected with pointers can be used to implement the data structures. Each structure contains data with one or more pointers to neighbouring structures.



There are several variants of the linked list structure:

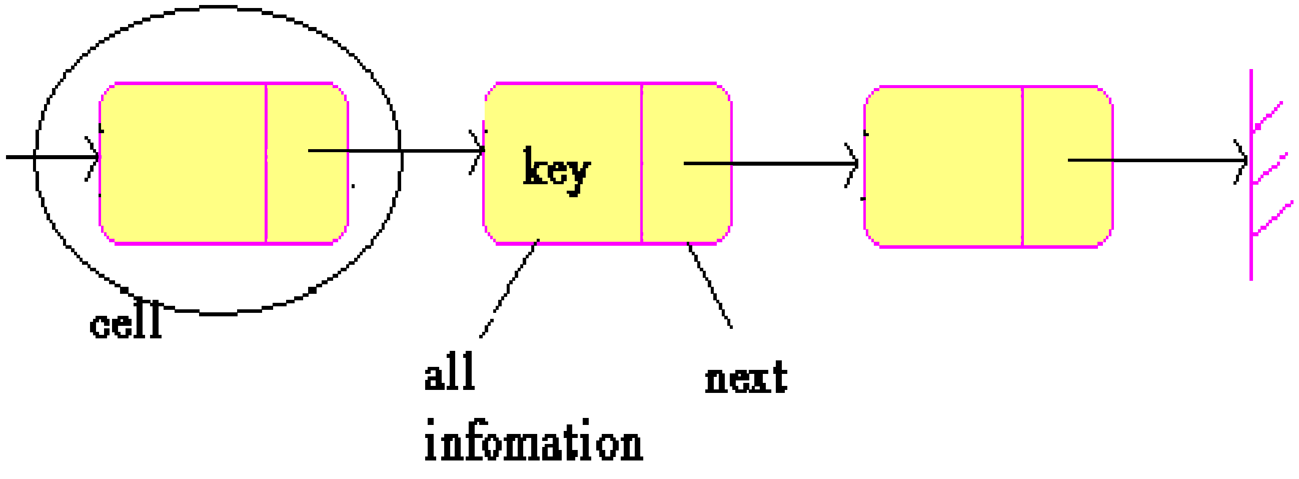
### Endogenous / Exogenous lists:

- endogenous lists have the data stored in the structure's KEY. The KEY is data stored within the structure.
- exogenous lists have the data stored outside the structure. Instead of a KEY, the structure has a pointer to the data in memory. Exogenous lists do

**not require data to be moved when individual cells are moved in a list; only the pointers to data must be changed. This can save considerable cost when dealing with large data in each cell. Another benefit of exogenous lists is many cells can point to the same data. Again, this may be useful depending on the application.**

**Here are example declarations of endogenous and exogenous structures:**

```
struct endogenous { data_type key,
      struct endogenous * next
      };
struct exogenous { data_type * data,
      struct exogenous * next
      };
```



**Circular / Non-circular lists:**

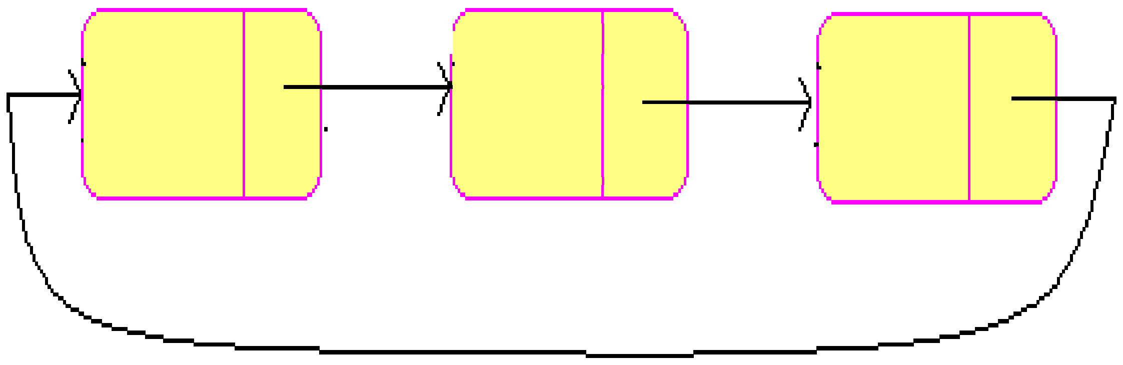
**- a circular list has the last cell in the array pointing to the first cell in the array. Specifically, the last cell's 'next' pointer references the first cell.**

**Representation in C : last_cell.next = &first_cell**

**- the last cell in a non-circular list points to nothing.**

**Representation in C : last_cell.next = NULL**

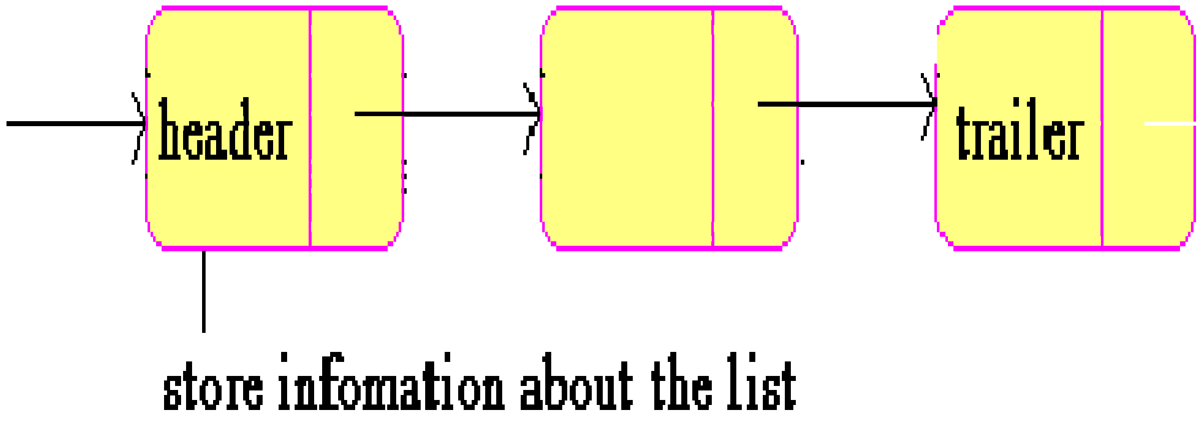
**Circular lists are useful for representing polygons (for example) because one can trace a path continously back to where one started. This is useful for representing a polygon because there is essentially no starting or ending point. Thus, we would like an implementation to illustrate this.**



**With/Without a Header/Trailer:**

**- a header node is a dummy first node in the list. It is not part of the data, but rather contains some information about the list (eg. size).**

**- a trailer node is at the end of a list (its contents marks the end).**

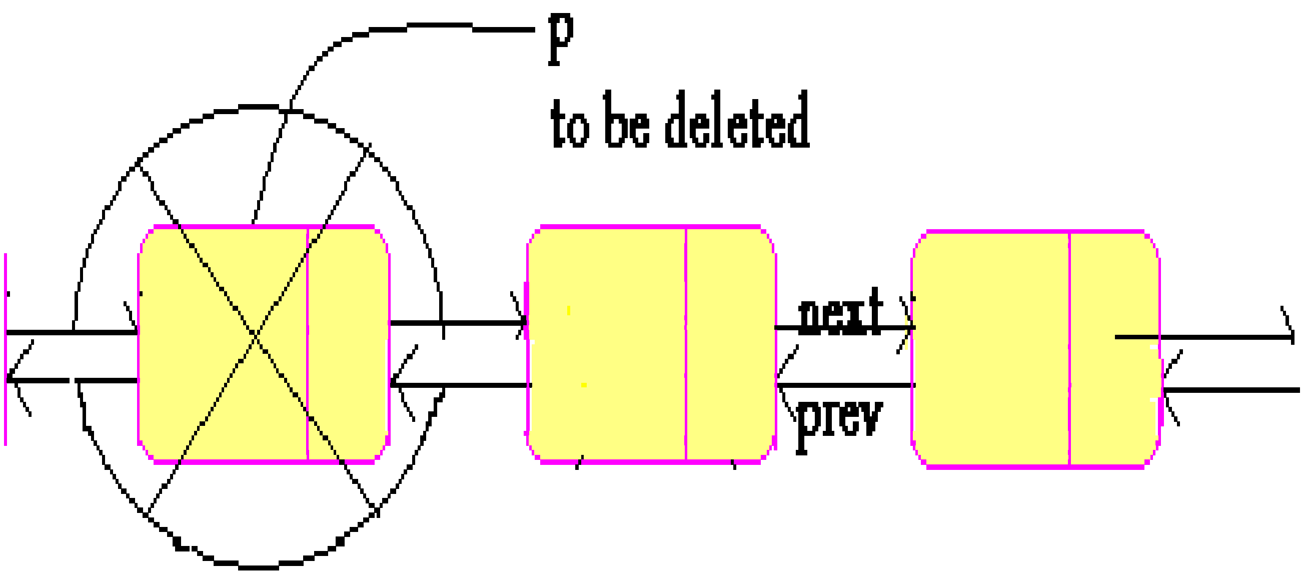


**Doubly Linked List:**

**-each node in a doubly linked list is a structure with two pointers to link with neighbouring nodes. One pointer points to the next node in the list, and the other pointer points to the previous node in the array. This implementation is useful for deleting nodes. The algorithm can be performed in O(1) time. Deleting nodes in a singly linked list can be done in OMEGA(n) time. Therefore, doubly linked lists can be very useful in applications requiring a lot of deletions. The pseudocode for the delete algorithm is as follows:**

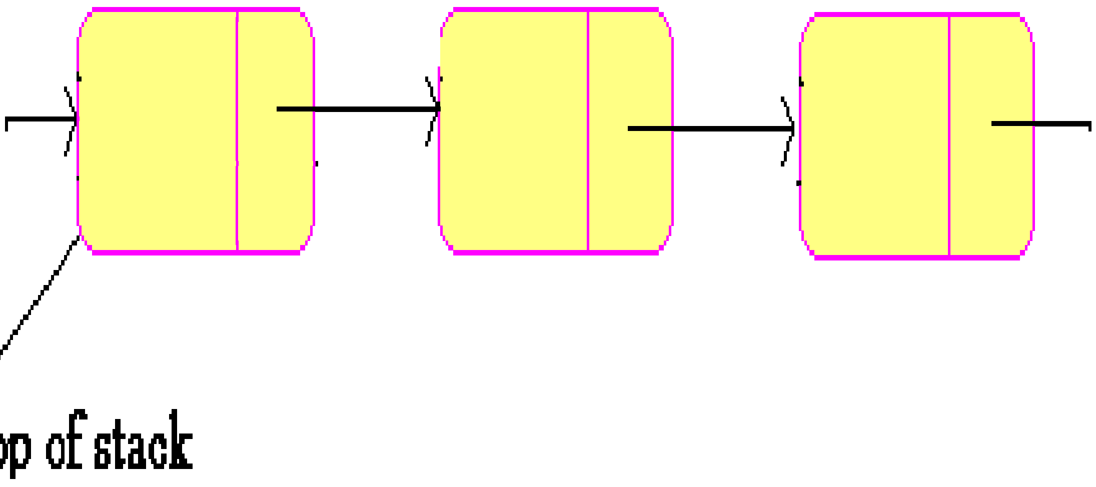
**p -> prev -> next = p -> next**

**p-> next -> prev = p -> prev**

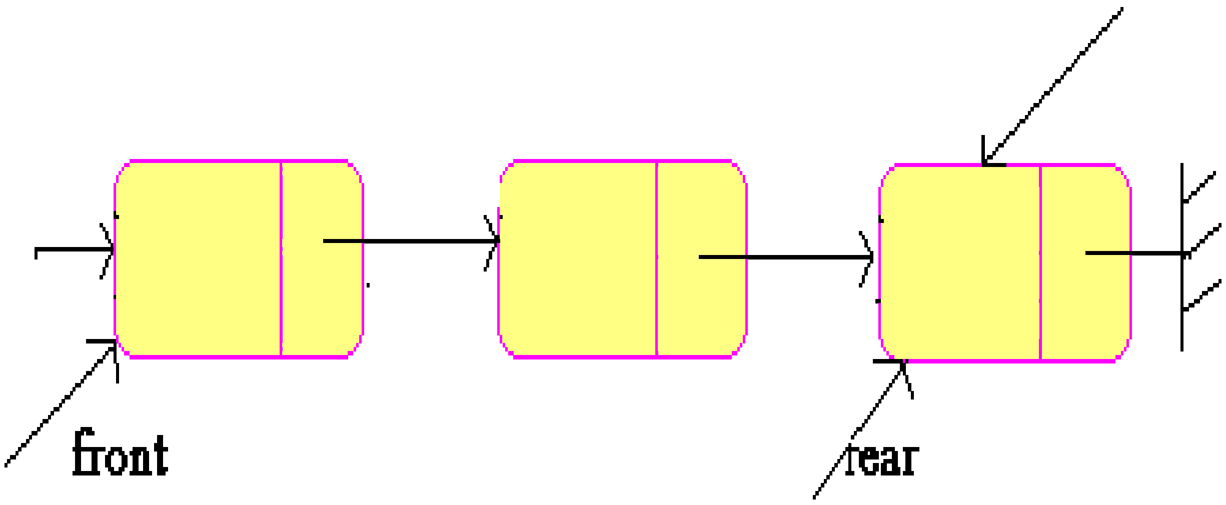


## 3. Utility of implementations:

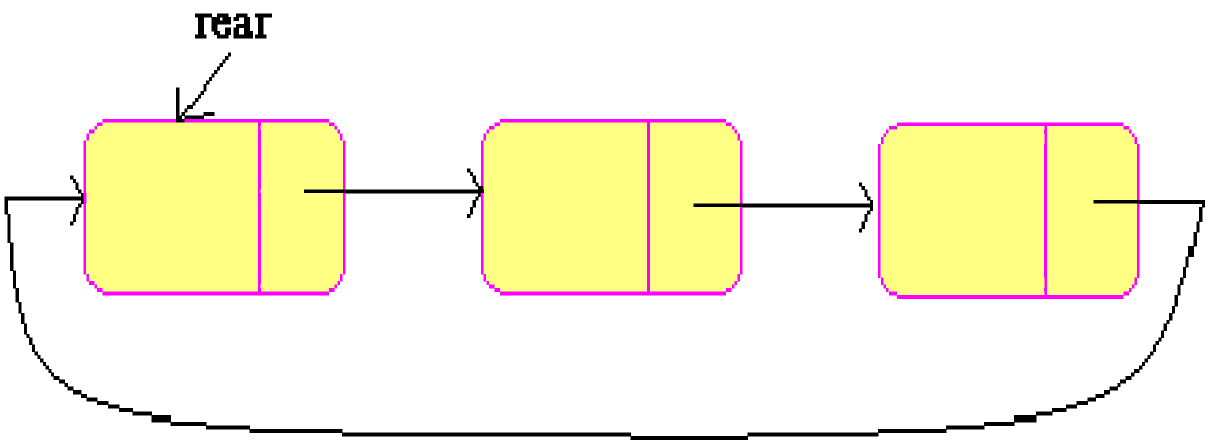
**-Stacks can easily be implemented as a linked list. One needs a pointer to the top of the stack (which is the front of the list).**



**-Queues can easily be implemented as a linked list as well. One needs a pointer to the front of the queue (which is the front of the list); as well, one needs a pointer to the rear of the queue (which is the end of the list).**



**It is better, however, to implement a queue as a circular list. This circular list has a pointer to the list pointing at the rear rather than the front of the list.**



**-Concatenating sublists can easily be done with a linked list. Conversely, concatenating sublists with an array implementation is expensive.**

**-Referencing sublists with a linked list implementation is expensive. Conversely, sublists can be referenced easily with an array implementation.**

---

# Applications

## 1. Polynomial ADT:

**A polynomial can be represented with primitive data structures. For example, a polynomial represented as $a_k x^k$ $a_{k-1} x^{k-1}$ + ... + $a_0$ can be represented as a linked list. Each node is a structure with two values: $a_i$ and i. Thus, the length of the list will be k. The first node will have ($a_k$, k), the second node will have ($a_{k-1}$, k-1) etc. The last node will be ($a_0$, 0).**

**The polynomial $3x^9 + 7x^3 + 5$ can be represented in a list as follows: (3,9) --> (7,3) --> (5,0) where each pair of integers represent a node, and the arrow represents a link to its neighbouring node.**

**Derivatives of polynomials can be easily computed by proceeding node by node. In our previous example the list after computing the derivative would represented as follows: (27,8)**

**--> (21,2). The specific polynomial ADT will define various operations, such as multiplication, addition, subtraction, derivative, integration etc. A polynomial ADT can be useful for symbolic computation as well.**

## 2. Large Integer ADT:

**Large integers can also be implemented with primitive data structures. To conform to our previous example, consider a large integer represented as a linked list. If we represent the integer as successive powers of 10, where the power of 10 increments by 3 and the coefficent is a three digit number, we can make computations on such numbers easier. For example, we can represent a very large number as follows:**

**$513(10^6) + 899(10^3) + 722(10^0)$.**

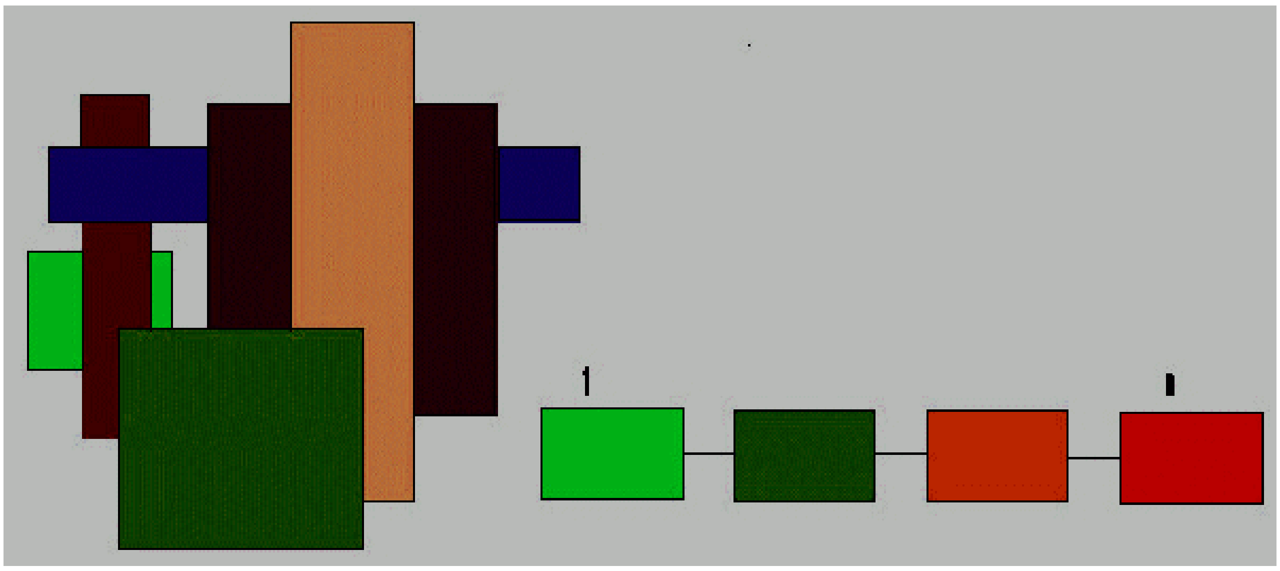
**Using this notation, the number can be represented as follows:**

**(513) --> (899) --> (722).**

The first number represents the coefficient of the $10^6$ term, the next number represents the coefficient of the $10^3$ term and so on. The arrows represent links to adjacent nodes. The specific ADT will define operations on this representation, such as addition, subtraction, multiplication, division, comparison, copy etc.

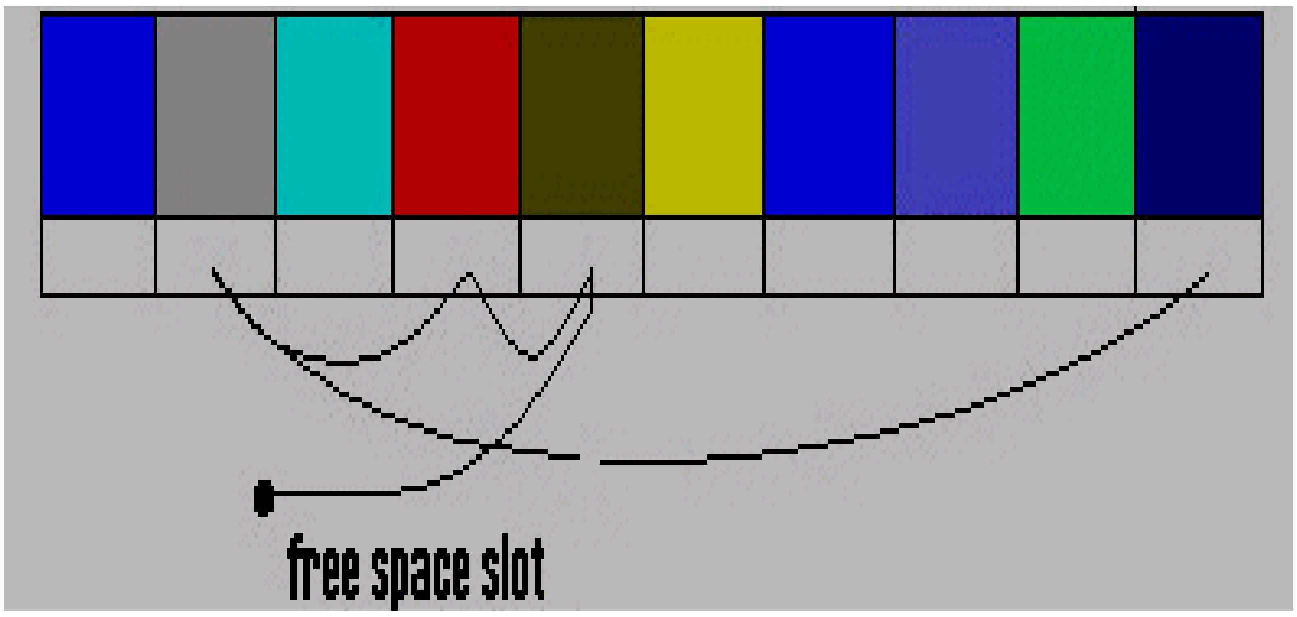
## 3. Window Manager ADT:

A window interface can be represented by lists. Consider an environment with many windows. The fist node in the list could represent the current active window. Subsequent windows are further along the list. In other words, the nth window corresponds to the nth node in the list. The ADT can define several functions, such as Find_first_window which would bring a window clicked upon to the front of the list (make it active). Other functions could perform window deletion or creation.



## 4. Management of free space:

**When memory is requested, a list of available blocks of memory might be useful. Again, a list could represent blocks in memory available to the user, with nodes containing pointers to these available blocks. The list can be used like a stack (LIFO). The last freed memory becomes the next available to the user. Such lists are called 'free space lists' or 'available space lists'. Since addition and deletion of nodes is at one end, these lists behave like stacks. All operations on free space lists can be done in O(1) time.**



# Stacks

## 1. Stack based languages:

**Expressions can be evaluated using a stack. Given an expression in a high-level language (for example, (a + b) * c) the compiler will transform this expression to postfix form. The postfix form of the above example is ab + c *, where a and b are operands, + and * are operators, and the expression is scanned left to right. The expression is pushed on the stack and evaluated as it is popped. The following algorithm illustrates the process:**

```
makenull(S)
y <- POP(S)
read(char)
if (char) is operand{
PUSH (char, S)
}
if (char) is operator{
x <- POP(S)
z <- evaluate "y char x"
PUSH (z, S)
}
```

**In the end the stack will hold one element (the result).**

## 2. Text Editor:

**A text editor can be implemented with a stack. Characters are pushed on a stack as the user enters text. Commands to delete one character or a command to delete a series of characters (for example, a sentence or a word) would also push a character on a stack. However, the character would be a unique identifier to know how many characters to delete. For example, an identifier to delete one character would pop the stack once. An identifier to delete a sentence would pop all characters until the stack is empty or a period is encountered.**

## 3. Postscript:

**Postscript is a full-fledged interpreted computer language in which all operations are done by accessing a stack. It is the language of choice for laser printers. For example, the Postscript section of code**
**1 2 3 4 5 6 ADD MUL SUB 7 ADD MUL ADD**
**represents:**
**1 2 3 4 5 6 + * - 7 + * +**
**which in turn represents:**
**1 + ( 2 * ( ( 3 - ( 4 * ( 5 + 6 ) ) ) + ( 7 ) ) )**
**Very much as in the stack-based language example, the expression can be evaluated from left to right. Expressions written in the form**

given above are called postfix expressions. Their easy evaluation with the help of a stack makes them natural candidates for the organization of expressions by compilers.

## 4. Scratch pad:

Stacks are used to write down instructions that you can not act on immediately. For example, future work to be done by the program, information that may be useful later, and so forth (just as with a scratch pad). An example of this is the rat-in-maze problem (see below).
A stack can be used to solve the problem of traversing a maze. One must keep track of previously explored routes, or else an infinite loop could occur. For example, with no previous knowledge of exploring a specific route unsuccessfully, one can enter a path, find no solution to the maze, exit the path through the same route as entrance, then enter the same unsuccessful path all over again. This problem can be solved with the help of a stack.
If we consider each step through a maze a cell, the following algorithm will traverse a maze successfully with the help of a stack 'S':
(For all cells)

```
Visited(cell) <- false
S <- Start
Visited(start) <-true
While not EMPTY(S) do{
if TOP(S) has an empty adjacent square then
<Q< (TOP(S))="EMPTY" SQUARE< DD>
VISITED(q) <- true
if q = 'TARGET CELL' then stop
PUSH(q, S) /*S has your path*/
else POP(S)
}
```

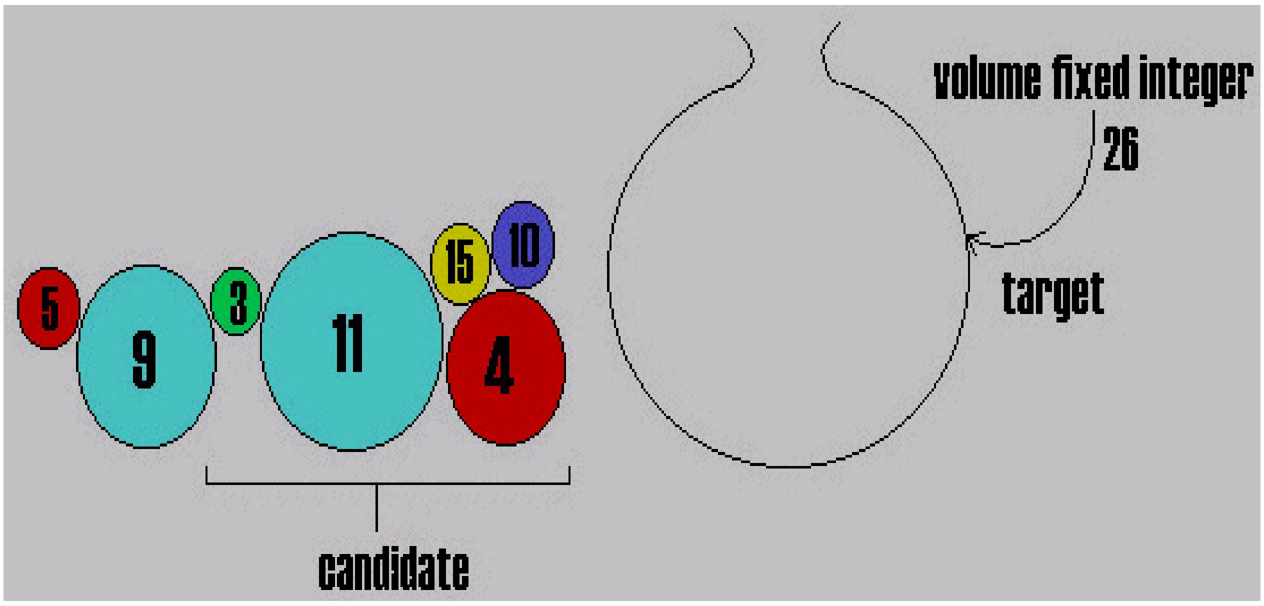
## 5. Recursion:

**Stacks are used in recursions. Every recursive program can be rewritten iteratively using a stack. One related problem is the knapsack problem:**
**Consider a knapsack with volume represented as a fixed integer. One is given a series of items of varying size (the size of the objects is represented as an integer). The knapsack problem is to find a combination of items that will fit exactly into the knapsack (i.e. no unused space). The function call is written as 'knapsack(target: , candidate: )' where 'target' is the amount of space left in the sack, and 'candidate' is the reference to the item being**

**considered to be added. The function returns a boolean result; 'true' if target can be filled exactly using a subest of the items numbered "candidate, ..., n". Here 'n' is the total number of items. Define size[.] as an array of sizes of the items. The following is a recursive solution to the problem:**

```
knapsack(target,candidate)
if target = 0 then return "true"
if candidate > n or target < 0 then return "false"
if knapsack(target - size(candidate) , candidate + 1) then
    return "true"
else return knapsack(target, candidate + 1)
```

**A knapsack of size 26 can be filled with items of size 15 and 11.**


volume fixed integer
26
target
5
9
3
11
15
10
4
candidate