

DATA STRUCTURES

References

- Langsam Y, Augenstein M J and Tanenbaum A M. *Data Structures using C and C++* Prentice Hall (1995); ISBN 0130369977.
- Aho A V, Hopcroft J F and Ullman J D. *Data Structures and Algorithms* Addison-Wesley (1983), chaps 2 – 3, 5, 8; ISBN 0201000237.
- Weiss M A. *Data Structures and Algorithm Analysis in C++* Addison-Wesley (2nd edition, 1999); ISBN 020161250X.
- Brunskill D and Turner J. *Understanding Algorithms and Data Structures* McGraw-Hill (1996); ISBN 0077091418.
- Tremblay and Sorenson. *An Introduction to Data Structures with Applications* McGraw-Hill (1983), chaps 3 – 5, 7; ISBN 0070651574.
- Knuth. *The Art of Computer Programming* Vol 1 “Fundamental Algorithms” Addison-Wesley, chap 2.

This course considers the types of structure best suited to particular operations on data, having regard to the *efficiency* of these operations and the effective use of *storage*.

Algorithms will be expressed in a C/C++/Java-like notation, though variations from these will be used. In most cases, type definitions, variable declarations and parameter types will be omitted, but should be evident from the context. In general, functions will not be called recursively; this is because we are studying the manipulation of data structures with particular emphasis on efficiency and are concerned with the underlying iterative processes of current machines. As a by-product, we will gain some understanding of the implementation of recursion in languages that permit it.

Since data structures may be considered as graphs, the term *node* (and also the term *record*) is often used for a C/C++ *structure* (i.e. a **struct**) or a C++/Java class with no member functions; the data members of a **struct** or class are often called the *fields* of the node or record. In our study of data structures, we will usually be interested in a node as a whole and in those fields of a node that contain pointers, but it is important to remember that the non-pointer data of a node may be quite complex.

Notation

We will sanitize some C/C++/Java notation, replacing it with more user friendly notation which is also more common outside the C/C++/Java world. In particular, we make the following replacements:

C/C++ notation	replacement
=	←
==	= /* take special note of this */
!=	≠
->	↑ /* just . in Java not -> */
NULL	nil /* null in Java */

LINEAR LISTS

To quote from Knuth:

a linear list is a set of n nodes (where $n \geq 0$) $X[1], X[2], X[3] \dots X[n]$ whose structural properties essentially involve only the linear (one-dimensional) relative positions of the nodes, i.e., the facts that:

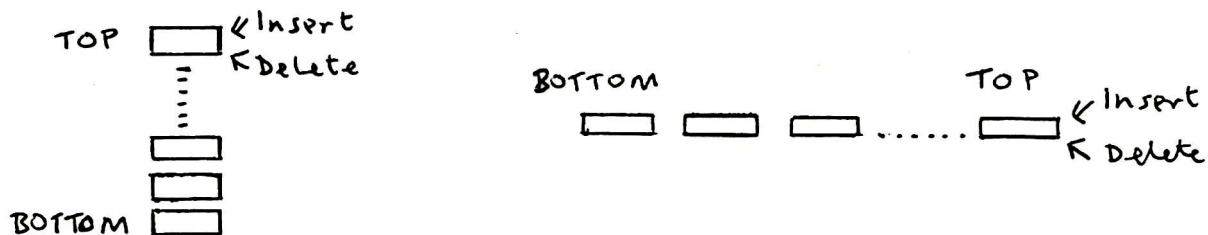
- if $n > 0$, $X[1]$ is the first node;
- when $1 < k < n$, the k th node $X[k]$ is preceded by $X[k-1]$ and followed by $X[k+1]$;
- $X[n]$ is the last node.

Examples of linear lists include: (1) one-dimensional arrays, (2) sequential files (eg on magnetic tape), (3) character strings, (4) linked lists.

There are many applications of linear lists in which *insertion* (i.e., addition) and *deletion* of nodes is required but in which this requirement is restricted to one end or both ends of the list. Often the requirement for *accessing* the nodes is similarly restricted. In such cases, we are interested in *three types of linear list*.

(1) STACK (or LIFO, push-down list, nesting store).

All deletions and insertions (and usually accesses) occur at one end of the list, known as the *top*.

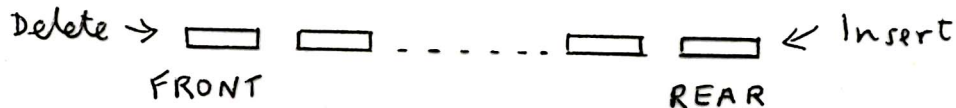


STACK(A,y) : Insert item y as a new top node on stack A .

UNSTACK(A,x) : Assign the top node of stack A to x and remove that top node from the stack.

(2) QUEUE (or FIFO)

All deletions (and usually accesses) occur at one end called the *front*, and all insertions at the other end called the *rear* of the list.

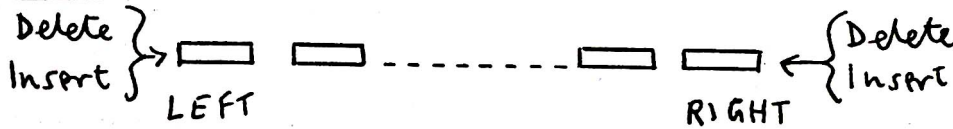


QUEUE(A,y) : Insert item y at the rear of queue A .

UNQUEUE(A,x) : Assign the node at the front of queue A to x , and remove that front node from the queue.

(3) DEQUE ('Double-Ended Queue')

With a *general deque*, deletions and insertions (and usually accesses) occur only at the *left* and *right* ends of the list.



With an *input-restricted deque*, deletions occur at both ends but insertions at only one end; with an *output-restricted deque*, insertions occur at both ends but deletions at only one end.

LEFTINSERT(A,y) : Insert item y on the left end of deque A.

RIGHTDELETE(A,x) : Assign the rightmost node of deque A to x and delete that node from the deque.

Likewise LEFTDELETE(A,x) and RIGHTINSERT(A,y)

We will consider the implementation of these three types of linear list in storage. First we look at their implementation in single (sequential, contiguous) areas of storage, and then at their implementation in linked lists – separate areas of storage linked by pointers.

Linear lists stored in sequential areas of memory

Nodes $X[1], X[2], X[3] \dots X[n]$ are stored in sequential areas of store, one after the other. We will assume that all nodes in a list are the same length: each occupies c bytes of memory. Then:

$$\begin{aligned} \text{Address}(X[j+1]) &= \text{Address}(X[j]) + c \\ \text{Address}(X[j]) &= \text{Address}(X[1]) + c(j-1) \end{aligned}$$

Now assume that cM consecutive bytes of store have been set aside for storing nodes $X[1]$ to $X[M]$. Consider the use of this area of store for the three structures noted above.

Note: We are not using $X[0]$.

(1) **STACK.** An integer variable T (sometimes called the “stack pointer”) gives the “subscript” of the top node of the stack. Initially $T = 0$, which denotes an empty stack. (It has been shown above that if $\text{Address}(X[1])$ is known, $\text{Address}(X[k])$ is easily determined for any k . Subscript notation is therefore used in the following examples, but remember that each reference to a subscripted variable involves, at machine level, at least an addition and (unless $c = 1$) also a multiplication.)

STACK(X,y) – Insert item y on stack X

```
if (T ≥ M) deal_with_OVERFLOW();
T ← T + 1;
X[T] ← y;
```

UNSTACK(X) – Return the top item on stack X and delete it from the stack

```
if (T = 0)
    deal_with_UNDERFLOW();
else
{   Result ← X[T];
    T ← T - 1;
    return Result;
}
```

(2) **QUEUE**. R gives the subscript of the rear node of the queue; F gives the subscript of the last node removed from the queue; N is the number of nodes in the queue. (The use of N is not strictly necessary but it makes things simpler.) Initially, $N = 0$ and $F = R = x$, where $0 < x \leq M$.

QUEUE(X,y) – Insert a new node in queue X

```

if (N = M) deal_with_OVERFLOW();
N ← N + 1;
R ← (R = M ? 1 : R + 1);
X[R] ← y;

```

UNQUEUE(X) – Return the front item of queue X and delete it from the queue

```

if (N = 0)
    deal_with_UNDERFLOW();
else
{
    N ← N - 1;
    F ← (F = M ? 1 : F + 1);
    return X[F];
}

```

(3) **GENERAL DEQUE**. The above algorithms will *insert at rear* and *delete at front*. It remains to specify:

Delete at rear

```

if (N = 0)
    deal_with_UNDERFLOW();
else
{
    N ← N - 1;
    Result ← X[R];
    R ← (R = 1 ? M : R - 1);
    return Result;
}

```

Insert at front

```

if (N = M) deal_with_OVERFLOW();
N ← N + 1;
X[F] ← y;
F ← (F = 1 ? M : F - 1);

```

If sequential allocation is used for queues and dequeues, OVERFLOW usually (but not invariably) results in the process being abandoned; but consider the case of *stacks* in more detail. If only *two* stacks coexist in store, the area of store remaining after allocation for programs and fixed data may be arranged so that the bottoms of the two stacks are at opposite ends of the area, and both stacks expand towards the middle; in this case, no overflow need occur until the complete area is exhausted. More complicated algorithms are required for more than two stacks; see Knuth (pp 243-248).