Jun25 180

Data types and structures

Primitive types

Boolean (true or false, 1 or 0)

Character

Integer, integral or fixed-precision values

Floating-point, single-precision real number values

Double, a wider floating-point size

Enumerated type, a small set of uniquely-named values

Composite types

Array

Record (also called tuple or struct)

Union

List

Stream

Set

Multiset

Stack

Queue

Double-ended queue

Tree

Graph

slide abt chars

An *enumerated type* (also called *enumeration*, *enum*, or *factor*) is a data type consisting of a set of named values called elements, members or enumerators of the type.

For example, an enumerated type called color may be defined to consist of the enumerators Red, Green, Zebra, Missing, and Bacon. In some languages, the declaration of an enumerated type also intentionally defines an ordering of its members while in others, the enumerators are unordered.

An *array* describes a collection of elements (values or variables) labeled by one or more indices (identifying keys) that can be computed at run time by the program. It is analogous to the mathematical concepts of vector and matrix. Array types with one and two indices are often called vector type and matrix type, respectively.

An *associative array* (also called a *map*, *symbol table*, or *dictionary*) is a data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection.

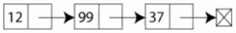
A *multimap* (sometimes also *multihash*) is a generalization of a map or associative array in which more than one value may be associated with and returned for a given key.

A *record* (also called *struct* or *compound data*) is a collection of fields, possibly of different data types, typically in fixed number and sequence. The fields of a record may also be called members or elements.

Records are distinguished from arrays by the fact that their number of fields is typically fixed, each field has a name, and that each field may have a different type.

A *list* or *sequence* represents an ordered sequence of values, where the same value may occur more than one time.

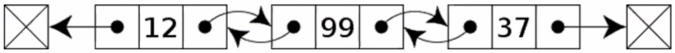
An example of a list is a computer representation of the mathematical concept of a finite sequence. The (potentially) infinite analog of a list is a *stream*.



A singly linked list structure, implementing a list with 3 integer elements.

The name lists are also used for several applications.

In computer science, a *doubly linked list* is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called links, which are references to the previous and to the next node in the sequence of nodes. The beginning and ending nodes' previous and next links, respectively, point to some kind of terminator, typically a sentinel node or null, to facilitate traversal of the list. If there is only one sentinel node, then the list is circularly linked via the sentinel node.



A doubly linked list whose nodes contain three fields: an integer value, the link to the next node, and the link to the previous node.

A *stream* is a sequence of data elements made available over time. A stream can be thought of as items on a conveyor belt being processed one at a time rather than in large batches.

A *set* is an abstract data type that can store certain values, without any particular order, and no repeated values. It is computer implementation of the mathematical concept of a finite set. Unlike most other collection types, rather than retrieving a specific element from a set, one typically tests a value for membership in a set.

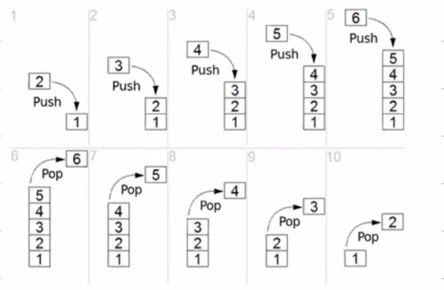
A generalization of the notion of a set is that of a *multiset* (or multiple membership set) or *bag*, which is similar to a set but allows repeated ("equal") values (duplicates of elements).

Other names for multisets:

aggregate, *bag*, *heap*, *bunch*, *sample*, *weighted set*, *occurrence set*, and *fireset* (finitely repeated element set)

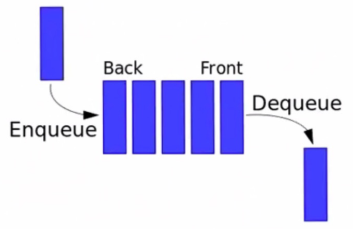
Indian mathmatician B — A ~ 12th century

A *stack* is a collection of elements, with two principal operations: **push**, which adds an element to the collection, and **pop**, which removes the most recently added element that was not yet removed. The order in which elements come off a stack gives rise to its alternative name, **LIFO** (for last in, first out). An additional operation may give access to the top without modifying the stack.



push and *pop* operations

A *queue* is a collection in which the entities are kept in order and the principal (or only) operations on the collection are the **addition** of entities to the rear terminal position, known as *enqueue*, and **removal** of entities from the front terminal position, known as *dequeue*. It has the alternative name FIFO (First-In-First-Out) (FIFO).



FIFO (first in, first out) queue

A *double-ended queue* (*dequeue*, often abbreviated to *deque*, pronounced *deck*) is an abstract d[redacted] type that generalizes a queue, in which elements can be added to or removed from either the front (head) or back (tail).

A *priority queue* is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority. If two elements have the same priority, they are served according to their order in the queue.

An *undirected graph* consists of a finite set of vertices or nodes or points, together with a set of unordered pairs of these vertices called edges, arcs or lines.

A *directed graph* consists of a finite set of vertices or nodes or points, together with a set of ordered pairs of these vertices called arrows, directed edges, directed arcs or directed lines.

A *labeled graph* data structure also associate to each edge some edge value, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).

A *tree* is defined recursively (locally) as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.

A tree is an undirected graph without cycles.

Problem solving with Computers

1. Algorithm construction
2. Program writing
3. Testing
4. Utilization

Algorithm construction

1. Cleaning the core of the problem
2. Algorithm design
3. Algorithm validation (checking/proving correctness)
4. Algorithm evaluation (checking/proving complexity, safety, security,….)

Note cleaning can change the problem.

Descriptions of many problems have a lot of details/features. Some of them are essential while others are extraneous. **Cleaning** means elimination of the extraneous details to preserve only essential details/features.

Essential for what:

1. For the given problem
2. For finding (good) solution
3. For good algorithm design

Essential for whom:

1. For the user
2. For the algorithm designer
3. For the programmer

Stable Matching Problem

David gale + Lloyd shapley

1962

men $\rightarrow$ women

n n

perfect matching

ranking

$m_1 : W_{i_1} > W_{i_2} > \dots > W_{i_n}$

$W_1 : m_{i_1} > \dots > m_{i_n}$

$$\left.\begin{array}{l} (m, w) \ \& \ (m', w') \\ m(w') > m(w) \ \& \ w'(m) > w'(m') \end{array}\right\} \text{instability}$$

(m, w')

Matching Algorithm

```
Start  $\forall m \in M$ & $\forall w \in W$ are free
While $\exists m$ (free (m) & m didnt propose to $\forall w$)
    Choose such m
    Let w ( m (w) > m (w') for $\forall w' \neq w$ & m did not propose to w)
    if w is free, then (m, w) $\in$ S matching set
Else (m', w) $\in$ S
      If w (m') > w (m)
      Else w (m) > w (m') then
      S: (S \ (m', w)) $\cup$ {(m, w)}
      & free (m')
      End if
  End if
End while
Output $S_f$
```

Proposition 1: There no more than n^2 loops in MA

$M \times W$

Pr 2: S_f is a stable matching

Proof by contradiction

Assume S_f is not stable

(m, w) & $(m', w') \in S_f$

$m(w') > m(w)$ & $w'(m) > w'(m')$

Case 1:

m did not prop to w'

then $m(w) > m(w')$ contradiction → impossible

Case 2:

m propose to w'

then $w'(m') > w'(m)$ contradiction → impossible

Pr 2 is proved

Pr 3: MA stops & S_f is a perfect match