# String/Character I/O

There are several approaches to reading in the text input required by many of these problems. Either you can:

- Repeatedly get single characters (perhaps using a library function like `getchar`);
- Repeatedly get strings (perhaps using a library function like `scanf`) and break them down into single characters.
- Read the entire line as a string (perhaps using a library function like `gets`), and then parsing it by accessing characters in the string.
- Perhaps more modern ways using streams are easier, perhaps not.

## Basic Data Types

Selecting the right data structure makes a tremendous difference in the organization and complexity of a given program.

Be aware of your basic structured data types (arrays, records, multidimensional arrays, enumerated types) and what they are used for.

Linked structures provide great *flexibility* in how memory is used, but are often unnecessary when the largest possible size structure is known in advance.

This is true for all or almost all of the problems in this book. Except for argument passing, pointers were not used in any of the example programs in the book.

Pointer structures are often more complex to work with and debug than arrays (KISS).

## Abstract Data Types

Thinking of data structures in terms of abstract data types provides a higher-order way to think about program organization.

Abstract data types are defined by the *operations* you want to perform on the data. The correct implementation (i.e. arrays or linked structures) is determined *after* you have defined the abstract data type.

Modern object-oriented languages like C++ and Java come with standard libraries of fundamental data structures.

These eliminate the need to reinvent the wheel, once you know the wheel has been invented.

# Queues and Stacks

Stacks and queues are containers where items are retrieved according to the order of insertion, independent of content.

*Stacks* maintain *last-in, first-out* order.

*Push(x,s)* – Insert item $x$ on top of stack $s$.

*Pop(s)* – Return (and remove) the top item of stack $s$.

*Initialize(s)* – Create an empty stack.

*Full(s), Empty(s)* – Test whether the stack can accept more pushes or pops, respectively.

Note that there is no element search operation defined on standard stacks and queues.

Applications include (1) processing parenthesized formulas (push on a "(", pop on ")") (2) recursive program calls (push on a procedure entry, pop on a procedure exit), and (3) depth-first traversals of graphs (push on discovering a vertex, pop on leaving it for the last time).

Also when the insertion order does not matter at all, since stacks are a very simple container.

## Queues

*Queues* maintain *first-in, first-out* order.

*Enqueue(x,q)* – Insert item $x$ at the back of queue $q$.

*Dequeue(q)* – Return (and remove) the front item from queue $q$

*Initialize(q), Full(q), Empty(q)* – Analogous to these operation on stacks.

Applications include (1) implementing buffers, (2) simulating waiting lines, and (3) representing card decks for shuffling.

Implementations include circular queues and linked lists.

## Dictionaries

Dictionaries permit content-based retrieval, unlike the position-based retrieval of stacks and queues.

*Insert(x,d)* – Insert item $x$ into dictionary $d$.

*Delete(x,d)* – Remove item $x$ (or the item pointed to by $x$) from dictionary $d$.

*Search(k,d)* – Return an item with key $k$ if one exists in dictionary $d$.

Classical dictionary implementations include (1) sorted arrays, (2) binary search trees, and (3) hash tables.

The correct implementation largely depends upon whether insertions and deletions will be performed.

Hash tables are often the right answer in practice, for reasons of simplicity and performance.

## Priority Queues

*Priority queues* are data structures on sets of items supporting three operations –

*Insert(x,p)* – Insert item $x$ into priority queue $p$.

*Maximum(p)* – Return the item with the largest key in priority queue $p$.

*ExtractMax(p)* – Return and remove the item with the largest key in $p$.

Priority queues are used to (1) to maintain schedules and calendars and (2) in sweepline geometric algorithms where operations go from left to right.

The most famous implementation of priority queues is the binary heap, but it is often simpler to maintain a sorted array, particularly if you will not be performing too many insertions.

# Sets

Sets (or more strictly speaking *subsets*) are unordered collections of elements drawn from a given universal set $U$.

*Member(x,S)* – Is an item $x$ an element of subset $S$?

*Union(A,B)* – Construct subset $A \cup B$ of all elements in subset $A$ or in subset $B$.

*Intersection(A,B)* – Construct subset $A \cap B$ of all elements in subset $A$ and in subset $B$.

*Insert(x,S), Delete(x,S)* – Insert/delete element $x$ into/from subset $S$.

Set data structures get distinguished from dictionaries because there is at least an implicit need to encode which elements from $U$ are *not* in the given subset. For sets of a large or unbounded universe, the obvious solution is representing a subset using a dictionary.

For sets drawn from small, unchanging universes, bit vectors provide a convenient representation. An $n$-bit vector or array can represent any subset $S$ from an $n$-element universe. Bit $i$ will be 1 iff $i \in S$. Element insertion and deletion operations simply flip the appropriate bit. Intersection and union are done by "and-ing" or "or-ing" the corresponding bits together.

Since only one bit is used per element, bit vectors can be space efficient for surprisingly large values of $|U|$. For example, an array of 1,000 standard four-byte integers can represent any subset on 32,000 elements.

# Object Libraries

A general library of abstract data types cannot really exist in C language because functions in C can't tell the type of their arguments. Thus we would have to define separate routines such as `push_int()` and `push_char()` for every possible data type.

However, C++ has been designed to support object libraries. In particular, the *Standard Template Library* provides implementations of all the data structures defined above and much more. Each data object must have the type of its elements fixed (i.e., templated) at compilation time, so

```
#include <stl.h>

  stack<int> S;
  stack<char> T;
```

declares two stacks with different element types.

Useful standard Java objects appear in the `java.util` package. Almost all of `java.util` is available on the judge.

Appropriate implementations of the basic data structures include —

| Data Structure | Abstract class | Concrete class | Methods |
|---|---|---|---|
| Stack | No interface | `Stack` | `pop, push, empt` |
| Queue | `List` | `ArrayList, LinkedList` | `add, remove, cl` |
| Dictionaries | `Map` | `HashMap, Hashtable` | `put, get, conta` |
| Priority Queue | `SortedMap` | `TreeMap` | `firstKey, last` |
| Sets | `Set` | `HashSet` | `add, remove, co` |

## Ranking and Unranking Functions

Whenever we can create a numerical *ranking* function and a dual *unranking* function which hold over a particular set of items $s \in S$, we can represent any item by its integer rank.

The key property is that $s = unrank(rank(s))$. Thus the ranking function can be thought of as a hash function without collisions.

One can define ranking/unranking functions for permutations (1 to $n!$), subsets (1 to $2^n$), and playing card (1 to 52).

We can use ranking/unranking functions to (1) generate all of the objects, (2) pick one at random, and (3) sort and compare them.

To rank and unrank playing cards, we order the card values (low to high) and note that there are four distinct cards of each value. Multiplication and division are the key to mapping them from 0 to 51:

```
#define NCARDS  52      /* number of cards */
#define NSUITS  4       /* number of suits */

char values[] = "23456789TJQKA";
char suits[] = "cdhs";

int rank_card(char value, char suit)
{
     int i,j;        /* counters */

     for (i=0; i<(NCARDS/NSUITS); i++)
          if (values[i]==value)
               for (j=0; j<NSUITS; j++)
                    if (suits[j]==suit)
                            return( i*NSUITS + j );

      printf("Warning: bad input value=%d, suit=%d\n",value
}

char suit(int card)
{
     return( suits[card % NSUITS] );
}

char value(int card)
{
     return( values[card/NSUITS] );
}
```

## Assigned Problems

110201 (Jolly Jumpers) – Does the distances between neighbors of a set of $n$ numerical steps realize all vaules 1 to $n-1$? What data structure should you use? What graphs (other than a path) allow such structures?

110204 (Crypt Kicker) – Decode an alphabet permutation-encrypted message using a dictionary of words. What constraints does the dictionary imply?

110205 (Stack 'em Up) – Rearrange a deck of cards according to a set of allowable shuffle operations. How do shuffles operate as rearrangement operations (permutations)? How good are the traditional perfect shuffles at mixing up a deck?

110208 (Yahtzee) – How do we assign dice roles to categories so as to maximize our score? Do we need to try all possibilities, or can we be more clever?