DATA STRUCTURES USING 'C'

Lecture No.02

Data Structures

What is Program

- A Set of Instructions
- Data Structures + Algorithms
- Data Structure = A Container stores Data
- Algoirthm = Logic + Control

Functions of Data Structures

- Add
  - Index
  - Key
  - Position
  - Priority
- Get
- Change
- Delete

Common Data Structures

- Array
- Stack
- Queue
- Linked List
- Tree
- Heap
- Hash Table
- Priority Queue

How many Algorithms?

- Countless

Algorithm Strategies

- Greedy
- Divide and Conquer
- Dynamic Programming
- Exhaustive Search

Which Data Structure or Algorithm is better?

- Must Meet Requirement
- High Performance
- Low RAM footprint
- Easy to implement
  - Encapsulated

Chapter 1 Basic Concepts

- Overview: System Life Cycle
- Algorithm Specification
- Data Abstraction
- Performance Analysis
- Performance Measurement

1.1 Overview: system life cycle (1/2)

- Good programmers regard large-scale computer programs as systems that contain many complex interacting parts.
- As systems, these programs undergo a development process called the *system life cycle*.

1.1 Overview (2/2)

- We consider this cycle as consisting of five phases.
  - Requirements
  - Analysis: bottom-up vs. top-down
  - Design: data objects and operations
  - Refinement and Coding
  - Verification
    - Program Proving
    - Testing
    - Debugging

1.2 Algorithm Specification (1/10)

- 1.2.1 Introduction
  - An *algorithm* is a finite set of instructions that accomplishes a particular task.
  - Criteria
    - input: zero or more quantities that are externally supplied
    - output: at least one quantity is produced
    - definiteness: clear and unambiguous
    - finiteness: terminate after a finite number of steps
    - effectiveness: instruction is basic enough to be carried out
  - A program does not have to satisfy the finiteness criteria.

1.2 Algorithm Specification (2/10)

- Representation
  - A natural language, like English or Chinese.
  - A graphic, like flowcharts.
  - A computer language, like C.
- Algorithms + Data structures = Programs [Niklus Wirth]
- Sequential search vs. Binary search

1.2 Algorithm Specification (3/10)

- **Example 1.1 [*Selection sort*]:**
  – From those integers that are currently unsorted, find the smallest and place it next in the sorted list.

i	[0]	[1]	[2]	[3]	[4]
-	30	10	50	40	20
0	10	30	50	40	20
1	10	20	40	50	30
2	10	20	30	40	50
3	10	20	30	40	50

```
for (i = 0; i < n; i++) {
  Examine list[i] to list[n-1] and suppose that the
  smallest integer is  at list[min];

  Interchange list[i] and list[min];
}
```

Program 1.1: Selection sort algorithm

1.2 (4/10)

- Program 1.3 contains a complete program which you may run on your computer

```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x,y,t) ((t) = (x), (x)= (y), (y) = (t))
void sort(int [],int); /*selection sort */
void main(void)
{
  int i,n;
  int list[MAX_SIZE];
  printf("Enter the number of numbers to generate: ");
  scanf("%d",&n);
  if( n < 1 || n > MAX_SIZE) {
    fprintf(stderr, "Improper value of n\n");
    exit(1);
  }
  for (i = 0; i < n; i++) {/*randomly generate numbers*/
    list[i] = rand() % 1000;
    printf("%d  ",list[i]);
  }
  sort(list,n);
  printf("\n Sorted array:\n ");
  for (i = 0; i < n; i++) /* print out sorted numbers */
    printf("%d  ",list[i]);
  printf("\n");
}
void sort(int list[],int n)
{
  int i, j, min, temp;
  for (i = 0; i < n-1; i++)  {
    min = i;
    for (j = i+1; j < n; j++)
      if (list[j] < list[min])
        min = j;
    SWAP(list[i],list[min],temp);
  }
}
```

Program 1.3: Selection sort

1.2 Algorithm Specification (5/10)

- **Example 1.2 [*Binary search*]:**

[0]	[1]	[2]	[3]	[4]	[5]	[6]
8	14	26	30	43	50	52

left	right	middle	list[middle]	:	searchnum
0	6	3	30	<	43
4	6	5	50	>	43
4	4	4	43	==	43
0	6	3	30	>	18
0	2	1	14	<	18
2	2	2	26	>	18
2	1	-			

- Searching a sorted list

```
while (there are more integers to check) {
   middle = (left + right) / 2;
   if (searchnum < list[middle])
      right = middle - 1;
   else if (searchnum == list[middle])
      return middle;
   else left = middle + 1;
}
```

```
int binsearch(int list[], int searchnum, int left, int right) {
/* search list[0] <= list[1] <= … <= list[n-1] for searchnum.
Return its position if found. Otherwise return -1 */
    int middle;
    while (left <= right) {
            middle = (left + right)/2;
            switch (COMPARE(list[middle], searchnum)) {
            case -1: left = middle + 1;
                        break;
            case 0 : return middle;
            case 1 : right = middle – 1;
            }
    }
    return -1;
}
```

***Program 1.6: Searching an ordered list**

1.2 Algorithm Specification (7/10)

- 1.2.2 Recursive algorithms
  - Beginning programmer view a function as something that is invoked (called) by another function
    - It executes its code and then returns control to the calling function.

1.2 Algorithm Specification (8/10)

- This perspective ignores the fact that functions can call themselves (*direct recursion*).
- They may call other functions that invoke the calling function again (*indirect recursion*).
  - extremely powerful
  - frequently allow us to express an otherwise complex process in very clear term
- We should express a recursive algorithm when the problem itself is defined recursively.

1.2 Algorithm Specification (9/10)

- Example 1.3 [*Binary search*]:

```
int binsearch(int list[], int searchnum, int left,
                                         int right)
{
/* search list[0] <= list[1] <=  · · ·  <= list[n-1] for
searchnum. Return its position if found. Otherwise
return -1 */
   int middle;
   if (left <= right) {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: return
            binsearch(list, searchnum, middle + 1, right);
         case 0 : return middle;
         case 1 : return
            binsearch(list, searchnum, left, middle - 1);
      }
   }
   return -1;
}
```

Program 1.7: Recursive implementation of binary search

1.2 (10/10)

- Example 1.4 [*Permutations*]:

```
void perm(char *list, int i, int n)
/* generate all the permutations of list[i] to list[n] */
{
   int j, temp;
   if (i == n) {
      for (j = 0; j <= n; j++)
         printf("%c", list[j]);
      printf("    ");
   }
   else {
   /* list[i] to list[n] has more than one permutation,
   generate these recursively */
      for (j = i; j <= n; j++) {
         SWAP(list[i],list[j],temp);
         perm(list,i+1,n);
         SWAP(list[i],list[j],temp);
      }
   }
}
```

Program 1.8: Recursive permutation generator

```
lv0 perm: i=0, n=2 abc
lv0 SWAP: i=0, j=0 abc
lv1 perm: i=1, n=2 abc
lv1 SWAP: i=1, j=1 abc
lv2 perm: i=2, n=2 abc
print: abc
lv1 SWAP: i=1, j=1 abc
lv1 SWAP: i=1, j=2 abc
lv2 perm: i=2, n=2 acb
print: acb
lv1 SWAP: i=1, j=2 acb
lv0 SWAP: i=0, j=0 abc
lv0 SWAP: i=0, j=1 abc
lv1 perm: i=1, n=2 bac
lv1 SWAP: i=1, j=1 bac
lv2 perm: i=2, n=2 bac
print: bac
lv1 SWAP: i=1, j=1 bac
lv1 SWAP: i=1, j=2 bac
lv2 perm: i=2, n=2 bca
print: bca
lv1 SWAP: i=1, j=2 bca
lv0 SWAP: i=0, j=1 bac
lv0 SWAP: i=0, j=2 abc
lv1 perm: i=1, n=2 cba
lv1 SWAP: i=1, j=1 cba
lv2 perm: i=2, n=2 cba
print: cba
lv1 SWAP: i=1, j=1 cba
lv1 SWAP: i=1, j=2 cba
lv2 perm: i=2, n=2 cab
print: cab
lv1 SWAP: i=1, j=2 cab
lv0 SWAP: i=0, j=2 cba
```

1.3 Data abstraction (1/4)

- Data Type
  A *data type* is a collection of *objects* and a set of *operations* that act on those objects.
  – For example, the data type int consists of the objects {0, +1, -1, +2, -2, …, INT_MAX, INT_MIN} and the operations +, -, *, /, and %.
- The data types of C
  – The basic data types: char, int, float and double
  – The group data types: array and struct
  – The pointer data type
  – The user-defined types

1.3 Data abstraction (2/4)

- Abstract Data Type
  - An *abstract data type(ADT)* is a data type that is organized in such a way that the specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operations.
  - We know what is does, but not necessarily how it will do it.

1.3 Data abstraction (3/4)

- Specification vs. Implementation
  - An ADT is implementation independent
  - Operation specification
    - function name
    - the types of arguments
    - the type of the results
  - The functions of a data type can be classify into several categories:
    - creator / constructor
    - transformers
    - observers / reporters

1.3 Data abstraction (4/4)

- Example 1.5 [*Abstract data type N*

structure *Natural_Number* is

objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (*INT_MAX*) on the computer

functions:

for all $x, y \in$ *Nat_Number*; *TRUE*, *FALSE* $\in$ *Boolean*

and where +, −, <, and == are the usual integer operations

Nat_No Zero( )	::=	0
Boolean Is_Zero(x)	::=	**if** (x) **return** *FALSE* **else return** *TRUE*
Nat_No Add(x, y)	::=	**if** (($x + y$) <= *INT_MAX*) **return** $x + y$ **else return** *INT_MAX*
Boolean Equal(x, y)	::=	**if** ($x == y$) **return** *TRUE* **else return** *FALSE*
Nat_No Successor(x)	::=	**if** ($x ==$ *INT_MAX*) **return** x **else return** $x + 1$
Nat_No Subtract(x, y)	::=	**if** ($x < y$) **return** 0 **else return** $x - y$

end *Natural_Number*

::= is defined as

Structure 1.1: Abstract data type *Natural_Number*