Data Structures

R. K. Ghosh

IIT Kanpur

Abstract Data Types

Simple Data Types

Definition of a Data Type

A data type consists of two things, (i) a universe or a class of elements and (ii) a set of operations (or an algebra like in the math).

- Universe: set of integers

  integers = $\{$-327678, ..., -1, 0, 1, ..., 32767$\}$
- Operations on **integers**: Integer arithmetic
  - 0: $\rightarrow$ integer (costant integer)
  - +: integer $\times$ integer $\rightarrow$ integer
  - -: integer $\times$ integer $\rightarrow$ integer
- Exceptions: operations that are not defined.
  - Division by 0
  - Overflow
- Other simple types are float, double, characters.

Compound Data Types

- Data types made out of simple data types such as an array of elements of some type t.
- An integer array is a compound type:
  array[1..100] of integer;
- Similarly a finite set n of records or a structures can be made out of pair the elements of two simple types, e.g.,

```
struct student {
       int slNo;
       char name[30];
};
```

- A compound type can be used every where a simple type can.

Abstract Data Types

Definition of ADT

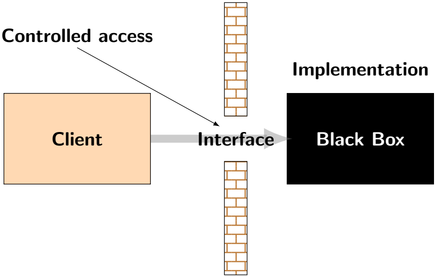
An Abstract Data Type (ADT) is an encapsulation of a data structure (implementation details of which are not visible to an outside client procedure) along with a collection of related operations on the encapsulated data structure.

- It is similar to a procedural or functional abstraction that deals with interface (a specification mechanism) rather than implementation of function or procedure.

Value of Abstraction

- Client uses abstraction and can only apply the operations exposed to outside. Internal implementations are hidden.
  - ADTs are best developed in Object Oriented Language like Java or C++.
  - Only public methods can be accessed by clients.
- Client cannot accidentally or intentionally modify internals.
  - Modification of variables can be prevented by declaring them as **private** or **protected**.
- ADTs provide a wall between implementor and user.
  - The wall effectively defines distinction of roles.
  - Interface (collection of public methods) is the way to talk.

Value of Abstraction



- Client declares initial object of type required.
  - An ordered list, a stack, a queue, a tree, etc.
- Client manipulates (accessor and modifier) through operation exposed to outside (provided as interface).
  - Push, pop, peek, enqueue, dequeue, etc.

Information Hiding

- In C, header files are used to separate out the implementation details.
- For example, a linked list is implemented as a self-referential structure.

```
#ifndef LIST_H
#define LIST_H
struct node {
   int info;
   struct node *next;
};
typedef struct node NODE;
```

List Operations

1. **newNODE()**: Allocating memory and create a new node.
2. **isEmpty(L)**: Find if L is empty.
3. **find(L, x)**: Given L and an element x returns ptr to node containing x if it exists.
4. **findNext(L, x)**: Returns ptr to node after node having x, if it exists.
5. **findPrevious(L, x)**: Returns ptr to node before the node having x if it exists.
6. **prepend(L, x)**: Insert x at the beginning of L.
7. **insert(L, x, y)**: Insert y after x in L.
8. **removeVal(L, x)**: Return ptr to L after deleting x if it exist.
9. **last(L)**: Returns ptr to the last element in L.

List Operations: newNODE, deleteList, isEmpty

```
NODE * deleteList(NODE * L) {
    return NULL;
};

NODE * newNODE() {
    NODE *p;
    p = (NODE *) malloc (sizeof(NODE));
    return p;
}

int isEmpty(NODE * L) {
    return (L == NULL);
}
```

List Operations: find

```
NODE * find(NODE * L, int x) {
    NODE *p;
    p = L;
    while (!isEmpty(p) && (p->info != x))
        p = p->next;
    return p;
}
```

List Operations: printList

```
void printList(NODE* L) {
   NODE * p;
   p = L;
   if (p == NULL) {
       printf("List is empty\n");
       return;
   }
   while (!isEmpty(p)) {
       printf("%d\t", p->info);
       p = p->next;
   }
   printf("\n");
   return;
}
```

List Operations: prepend

```
NODE * prepend(NODE *L, int x){
    NODE *p;
    if (isEmpty(p)) {
        p = newNODE();
        p->info= x;
        return;
    }
    p = newNODE();
    p->info = x;
    p->next = L;
    printf("List after insertion of %d: ", x);
    printList(p);
    return p;
}
```

List Operations: insert

```
void insert(NODE *L, int x, int y){
    NODE *p;
    NODE *q;
    p = find(L,x);
    if (p != NULL) {
        q = newNODE();
        q->info = y;
        q->next = p->next;
        p->next = q;
        printf("Insert after %d is successful\n", x)
            ;
        return;
    }
    printf("Insert %d failed: %d not found in list\
        n",y,x);
    return;
}
```

List Operations: findNext

```
NODE * findNext(NODE * L, int x) {
    NODE * p;
    NODE * q;
    if (isEmpty(L))
        return NULL;
    p = L;
    while (!isEmpty(p) && (p->info != x))
        p = p->next;
    if (p != NULL)
       return p->next;
    else
       return p;
}
```

List Operations: findPrevious

```
NODE * findPrevious(NODE * L, int x) {
    NODE * p;
    NODE * q;
    if (isEmpty(L))
        return NULL;
    p = L;
    while (!isEmpty(p) && (p->info != x)) {
        q = p;
        p = p->next;
    }
    if (p != NULL)
        return q;
    else
        return p;
}
```

```
void removeVal(NODE * L, int x) {
    NODE * p;
    p = findPrevious(L, x);
    if (p!=NULL)
        p->next = p->next->next;
    else
        printf("Delete failed: %d not in list\n", x
            );
    return;
};
```

List Operation: last

```
NODE * last(NODE * L){
    NODE * p;
    p = L;
    if (isEmpty(p)) {
        printf("List is empty\n");
        return NULL;
    }
    while (p->next != NULL)
        p = p->next;
    return p;
};
#endif
```

Arrays as ADTs

- Universe: set of all arrays with element type t.
- Array is a compound type.
- Operations:
  - Create a new array.
  - Get an element given a position.
  - Modify value of at a given position.
- Exceptions: Invalid operations.

Universe, Operations, Exceptions

```
// Universe: set of all arrays with element type t

// Operations:
new_t: integer × integer → array_t;
get_t: array_t × integer → integer;
put_t: array_t × integer × t → array_t;

// Exceptions:
get_t(new_t(1,10),20)  // Out of bounds
get_t(new_t(1,10),2)  // Uninitialized
```

Record as ADTs

record *// Universe of records*
    id$_1$: t_1 , **id**$_2$: t_2 , $\ldots$, **id**$_k$: t_k ,
end;

// Operations
new$_{t_1 \ldots t_k}$: $\rightarrow$ **record**$_{t_1 \ldots t_k}$ *// Creating a new record*
getid$_i$: **record**$_{t_1 \ldots t_k}$ $\rightarrow t_i$ *// Accessor method*
setid$_i$: **record**$_{t_1 \ldots t_k}$ $\times\ t_i \rightarrow$ **record**$_{t_1 \ldots t_k}$ *// Modifier method*

Stacks as ADTs

```
// Universe set of all stacks of type t
new_t :  → stack_t // Creating a stack
empty_t : stack_t → boolean
pop_t : stack_t → t × stack_t // Modifier
push_t : stack_t × t → stack_t // Modifier

// Exception: out of space
```

Stack: Declaration & Creation

```
typedef struct stack{
    int *info;
    int top;
    int limit;
} STACK;

void createStack(STACK *s, int maxSize) {
    s->info= (int *) malloc(maxSize*sizeof(int));
    if (s->info  == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    s->top = -1;
    s->limit = maxSize-1;
}
```

Stack: isEmpty, isFull and Top

```
int isEmpty(STACK *s) {
    return s->top == -1;
}

int isFull(STACK *s) {
    return s->top == s->limit;
}

int Top(STACK *s) {
    if (isEmpty(s)) {
        printf("Stack is empty\n");
        return -1;
    }
    return s->info[s->top];
}
```

Stack: Push and Pop

```
void push(STACK *s, int element) {
    if (isFull(s)) {
        printf("Stack full: insertion denied\n");
        return;
    }
    s->info[++s->top] = element; // pointer to end
        of list
    return;
}

int pop(STACK *s) {
    if (isEmpty(s)) {
        printf("Stack empty: deletion denied\n");
        return -999;
    }
    return (s->info[s->top--]);
}
```

Stack: Print

```
void printStack(STACK *s) {
    int i;
    if(isEmpty(s)) {
        printf("Stack is empty\n");
        return;
    }
    i = -1;
    while (i < s->top) {
        printf("%d\t", s->info[++i]); // print
            elements
    }
    printf("\n");
    return;
}
```

Assignment # 1

Questions (Full Marks 50)

1. Design ADT for the universe of n (constant) digit natural number with operations (constant, succ, addition multiplication, etc.) to support arithmetics involving n digit natural numbers using arrays and linked list ADTs. Define exceptions if any. [15]
2. Implement arithmetics of n digit natural numbers in C. You must have separate header files for implementing the operations and expceptions. [10]
3. Design ADT for Queues and sequence. [10]
4. Implement both queue and sequence using ADTs you have defined. Use sequence ADT for sorting. [15]

Assignment Rubric

- All handwritten assignment should be prepared using LaTeX.
- Programs and handwritten assignment will be checked for match, and if found, penalties will be as follows.
  - For the first instance, zero will be awarded for the assignment and 5% haircut applied to total marks.
  - For the 2nd instance, zero for the assignment, and one grade lower.
  - For multiple instances, zero for all assignments and one grade lower.
- Programming assignments would require demo outside class hours according to convenience of TAs.