Data Structures

R. K. Ghosh

IIT Bhilai

Abstract Data Types

Data Types

Simple Data Types

- Integers, characters, floats, double.

Definition of a Data Type

A data type consists of two things:

1. A universe or a class of elements, and
2. A set of operations (or an algebra)

Integer Data Types

- Universe: **integers** = {-32768, . . ., -1, 0, 1, . . ., 32767} - short integer (size 2B) in C.
- Operations on **integers**: Integer arithmetic

0	$\rightarrow$ **const integer**
+	**integer** $\times$ **integer** $\rightarrow$ **integer**
-	**integer** $\times$ **integer** $\rightarrow$ **integer**
$\times$	**integer** $\times$ **integer** $\rightarrow$ **integer**
/	**integer** $\times$ **integer** $\rightarrow$ **integer**

- Exceptions: raised for undefined operations.
  - Division by 0, overflow

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 RollNo;
        char name[30];
};
```

- A compound type can be used much like a simple type.

Abstract Data Types

Definition of ADT

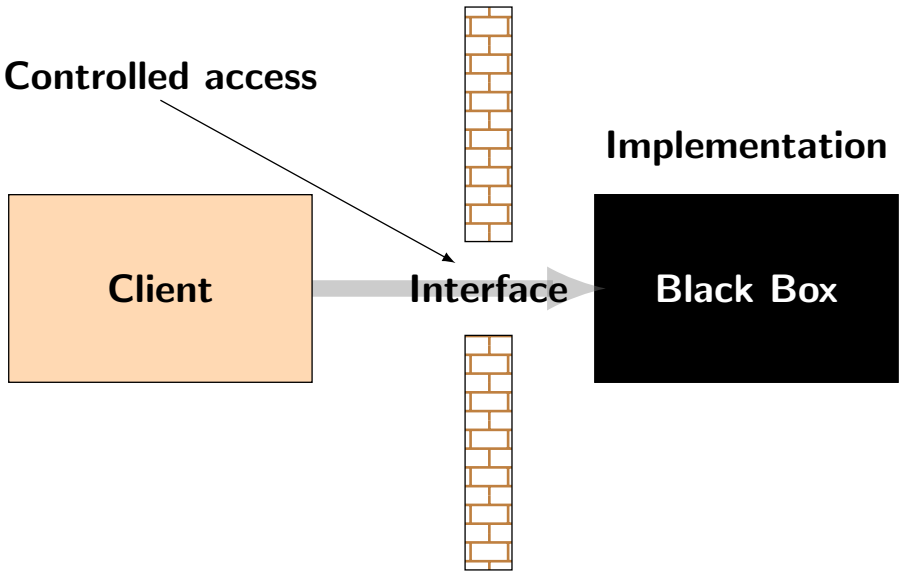
An Abstract Data Type (ADT) defines an encapsulation of a data structure **independent** of implementation details together with a set of operations on the encapsulated data structure.

- Abstraction deals with generality rather than specificity.
- Implementation or concreteness is not important but focus is on properties (data and functionality).
- So, ADT deals with interface (a mechanism for specification).

Value of Abstraction

- A client uses abstraction and applies the operations exposed to outside for manipulating a data structure.
  - Object Oriented Language like Java or C++ are ideal for implementing ADTs.
  - Public methods can be accessed by clients.
  - Client cannot accidentally or intentionally modify internals
  - Variables **private** or **protected** are mutation resistant.

Value of Abstraction



- Client declares initial object of the required type.
- Manipulates (accessor and modifier) through operation exposed to outside (provided as interface).
  - Push, pop, peek, enqueue, dequeue, etc.

Wall of ADT

An ADT is a wall between the implementor and the user. Defines separation of roles. User has to use public interface (methods) is to manipulate data structure.

Universe, Operations, Exceptions

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

// Operations:
new_T : integer × integer → array_T ;
get_T : array_T × integer → T ;
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*
fld$_1$: T_1 , **fld**$_2$: T_2 , ..., **fld**$_k$: T_k ,
end;

// Operations
new$_{T_1 \ldots T_k}$: $\rightarrow$ **record**$_{T_1 \ldots T_k}$ *// Creating a new record*
getfld$_i$: **record**$_{T_1 \ldots T_k}$ $\rightarrow$ T_i *// Accessor method*
setfld$_i$: **record**$_{T_1 \ldots T_k}$ $\times$ T_i $\rightarrow$ **record**$_{T_1 \ldots T_k}$ *// Modifier method*

Information Hiding

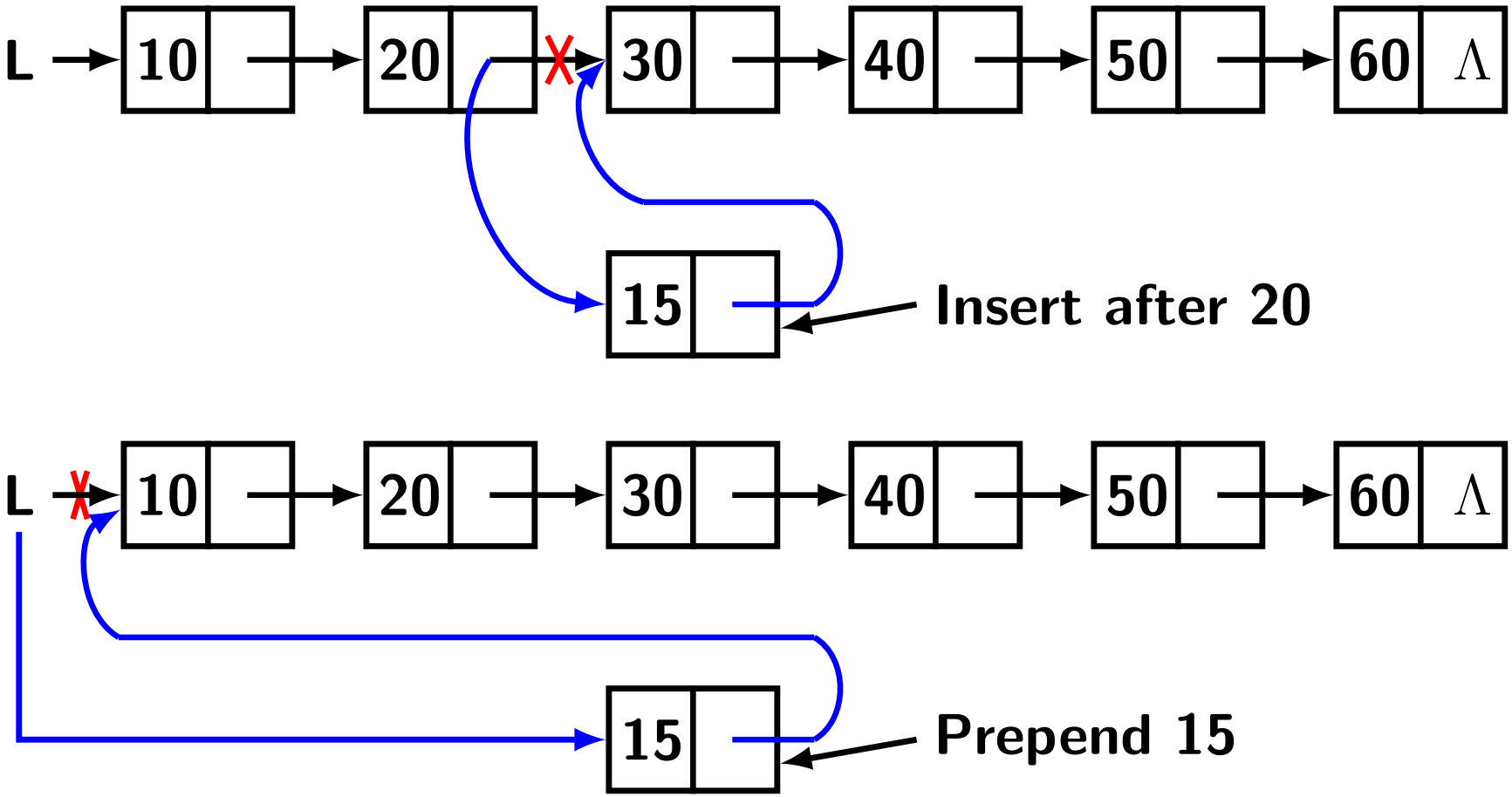
- In C, header files are used to separate out the implementation details.
- For example, a linked list is implemented as a self-referential structure.

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

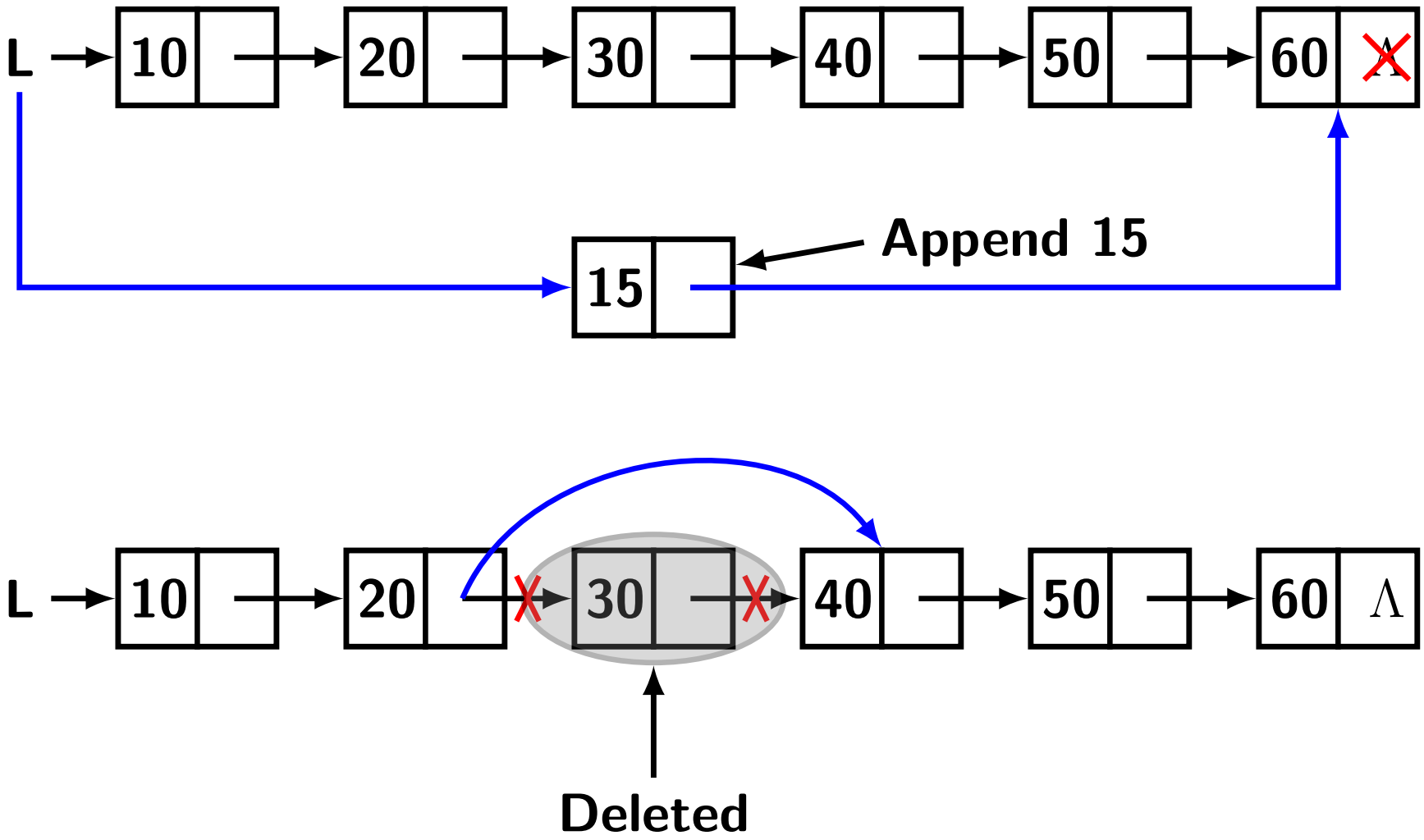
List Operations

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. **append(L, x)**: Insert x at the beginning of L.
8. **insert(L, x, y)**: Insert y after x in L.
9. **remove(L, x)**: Return ptr to L after deleting x if it exist.
10. **last(L)**: Returns ptr to the last element in L.

List Insertion



List Append



List Operations: NewNODE, NewList, IsEmpty

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

NODE * newList() {
    NODE *p;
    p = newNode(); // header node
    p->info = -1;
    p->next = NULL;
    return p;
}

int isEmpty(NODE * list) {
    return (list->next == NULL);
}
```

List Operations: FirstNode, LastNode

```
NODE *firstNode(NODE *list) {
    return list->next;
}

NODE * lastNode(NODE *list) {
    NODE *p = list;

    if (isEmpty(list))
        return p;
    while (p->next != NULL)
        p = p->next;
    return p;
}
```

List Operations: Find

```
NODE * find(NODE *list, int x) {
    NODE *p;

    p = firstNode(list);
    if (p != NULL) {
        while (p->info != x && p->next != NULL
            ) {
            p = p->next;
        }
    }
    return p;
}
```

List Operations: FindPred

```
NODE * findPred(NODE *list, int x) {
    NODE *p, *q;
    q = list;
    p = q->next;
    while (p->info != x && p->next != NULL) {
        q = p;
        p = p->next;
    }
    return q;
}
```

List Operations: FindSucc

```
NODE * findSucc(NODE *list, int x) {
    NODE *p;
    p = list;
    if (isEmpty(list)) {
        return p;
    }
    while (p->info != x) {
        p = p->next;
    }
    if (p != NULL)
        p = p->next;
    return p;
}
```

List Operations: PrintList

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

List Operations: Prepend

```
void prepend(NODE *list, int x) {
    NODE *p, *q;

    p = newNode();
    p->info = x;
    q = list;
    p->next = q->next;
    q->next = p;
    return;
}
```

List Operations: Append

```
void append(NODE *list, int x) {
    NODE *p, *q;

    p = newNode();
    p->info = x;
    p->next = NULL;

    q = lastNode(list);
    q->next = p;
    return;
}
```

List Operations: InsertBefore

```
void insertBefore(NODE *list, int x, int y) {
    NODE *p, *q, *r;

    q = findPred(list,x);
    if (q == NULL) {
        printf("Insert Before failed: %d not found\
            n",x);
        return;
    }
    p = newNode();
    p->info = y;
    p->next = q->next;
    q->next = p;
    return;
}
```

List Operations: InsertAfter

```
void insertAfter(NODE * list, int x, int y) {
    NODE *p, *q;

    q = find(list, x);
    if (q != NULL) {
        p = newNode();
        p->info = y;
        p->next = q->next;
        q->next = p;
    }
    return;
}
```

List Operations: Remove

```
void removeX(NODE *list, int x) {
    NODE *p, *q;

    p = findPred(list, x);
    q = findSucc(list, x);
    if (p != NULL)
        p->next = q;
    else
        printf("Error: %d is not present, remove
            failed\n", x);
    return;
}
```

List Operations: Length

```
int length(NODE * list) {
    NODE * p;
    int len = 0;
    if (isEmpty(list))
        return 0;
    p = list->next;
    while (p != NULL) {
        len++;
        p = p->next;
    }
    return len;
}
```

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.

Stacks as ADTs

// Universe set of all stacks of type T
new$_T$: $\rightarrow$ **stack**$_T$ *// Creating a stack*
empty$_T$: **stack**$_T$ $\rightarrow$ **boolean**
full$_T$: **stack**$_T$ $\rightarrow$ **boolean**
pop$_T$: **stack**$_T$ $\rightarrow T \times$ **stack**$_T$ *// Modifier*
push$_T$: **stack**$_T$ $\times T \rightarrow$ **stack**$_T$ *// Modifier*

// Exceptions: underflow and overflow
pop$_T$(S) : **empty**$_T$(S) == TRUE
push$_T$(S) : **full**$_T$(S) == TRUE

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;
}
```