Lists

Intro

A list is a data structure based on usage of pointers and dynamic allocation of memory.

With respect to other ADT (like arrays), a list:

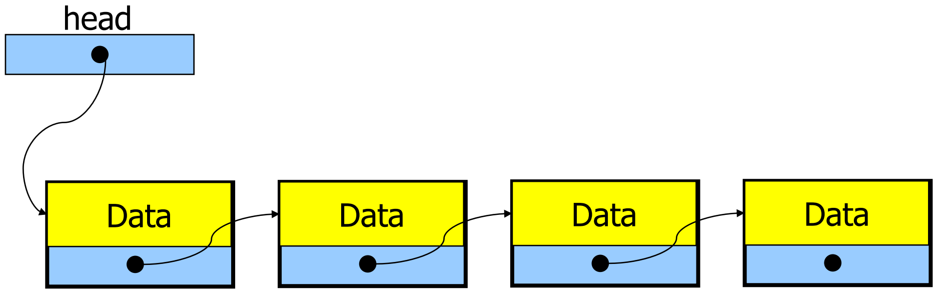
- provides more flexibility in memory usage
- but it is less efficient.

Definition

A list is an ADT where:

- Each element is allocated/deallocated separately
- Each element is *linked* to the others and accessible through pointers
- There is a variable (called head) which refers to the first element.

Simple List



Linked List Properties

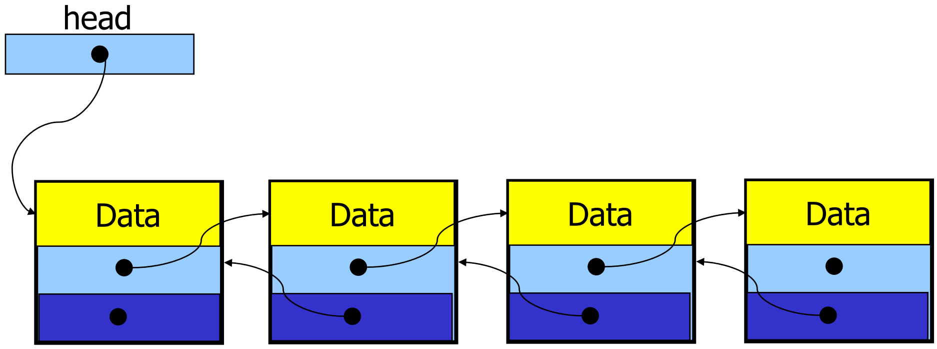
- In every moment only the necessary memory is effectively used
- Accessing to an element may require to access in sequence to all elements of the list

Variants

Other variants of linked lists, are:

- Circular lists, where last element points to the first element (the head of the list)
- Double-pointer Lists where each element contains both a pointer to the previous element and a pointer to the following element.

Double Pointer Lists



7

Implementation in C

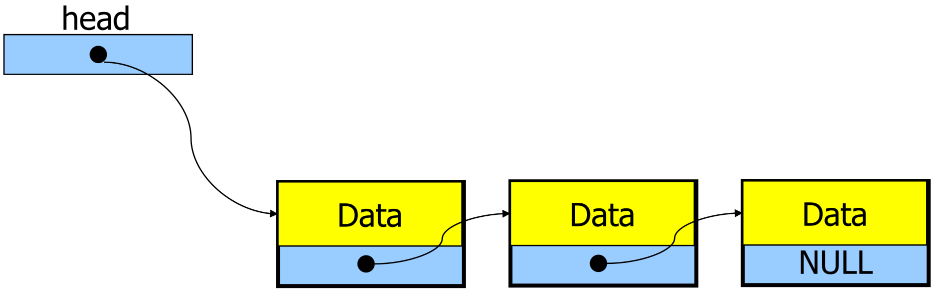
```
/* element definition */
typedef struct list_el{
     int code;
     char *name;
     char *surname;
     struct list_el * next;
} LIST_ELEMENT;

/* head pointer definition */
LIST_ELEMENT * head = NULL;
```

8

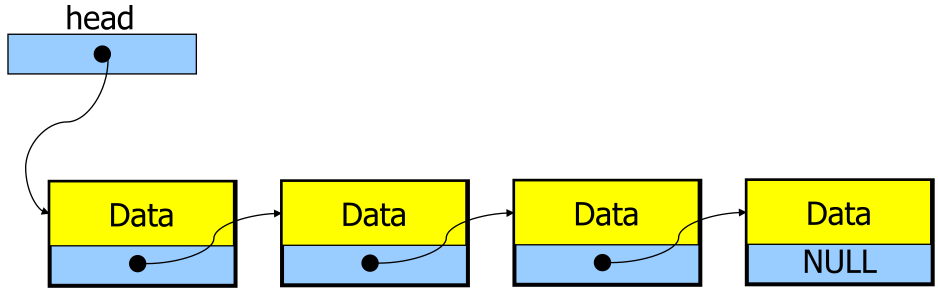
Head Insertion

```
…
LIST_ELEMENT * p;
…
p=(LIST_ELEMENT *) malloc(sizeof(LIST_ELEMENT));
if  (p==NULL)
  ERROR
p->code = val;
p->name = strdup(name);
p->surname = strdup(surname);
p->next = head;
head = p;
…
```

Before Head Insertion



After Head Insertion



Insert function

```
int insert (LIST_ELEMENT ** t, int val, char *name,
                                                   char *surname)
{  LIST_ELEMENT *p;
   p=(LIST_ELEMENT *) malloc(sizeof(LIST_ELEMENT));
   if (p==NULL)
        return (-1);
   p->codee=val;
   p->name = strdup(name);
   p->surname = strdup(surname);
   p->next=*t; /* head is *t   */
   *t=p;
   return 0;
}
```

Caller Program

```
…
LIST_ELEMENT * head;
int ret, val;
char name[MAX], surname[MAX];
…
scanf ("%d, %s %s\n", &val, name, surname);
ret=insert ( &head, val, name, surname);
if (ret == -1)
      ERRORE
```

Search

```
LIST_ELEMENT * search (LIST_ELEMENT *t, int val)
{  LIST_ELEMENT *p;
   p=t;
   while (p != NULL)
   {  if (p->code == val)
         return (p);
      p = p->next;
   }
   return  p;
}
```

Caller Program

```
…
LIST_ELEMENT *head, *p;
int val;
…
scanf ("%d\n", &val);
p= search (head, val);
if (p == NULL)
   printf ("Element not found \n");
else
   printf ("%d %s %s\n", p->code, p->name, p->surname);
…
```

Sorted Lists

If the insertion procedure puts the new element in the right position, then the list can be kept sorted (with respect to one field of the struct).

In this way is possible to:

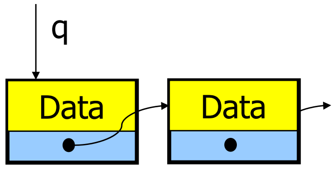
- Simplify search operations
- Access to elements in an ordered way.

Insertion in a sorted list

```
int insert_sorted (LIST_ELEMENT  **t, int val, char *name,
                                                   char *surname)
{   LIST_ELEMENT *p, *q;
   /* allocate new element */
   p= (LIST_ELEMENT *) malloc(sizeof(LIST_ELEMENT));
   if (p==NULL)
        return (-1);
   p->code=val;
   p->name = strdup(name);
   p->surname = strdup(surname);
   q = *t;
   /* head insertion */
   if( (q == NULL) || (q->code > val))
   {  p->next = *t;
      *t = p;
      return 0;
   }
```

Insertion in a sorted list

```
   /* insertion in the middle of the list */

   while( q->next != NULL)
    {   if( q->next->code   > val)
      {  p->next =  q->next;
         q->next = p;
         return 0;
      }
      q = q->next;
    }
   /* insertion in the end of the list */
   p->next = NULL;
    q->next = p;
   return 0;
}
```



Delete

Deleting an element usually requires:

- The search operation, which produces a pointer to the element to be canceled
- The actual delete operation which requires:
  - the pointer to the element to be canceled
  - and the pointer to the preceding element.

Delete

```
int delete(LIST_ELEMENT **t, int val)
{ LIST_ELEMENT *p, *q;
  q = *t;
  if (q==NULL) /* empty list */
    return (-1);
   /* head delete */
   if (q->code == val))
   {free (q->name);
    free (q->surname);
    *t = q->next;
    free (q);
    return 0;
  }
```

Delete

```
/* delete in the middle or in the end of list */
  while( q->next!= NULL)
   {  if( q-> next->code == val)
    { q->next = q->next->next;
      free (q->next->name);
      free (q->next->surname);
      free (q->next);
      return 0;
    }
    q = q->next;
   }
}
```

Complexity

Insertion in the head of the list has a complexity O(1).

All other operations on lists have O(n) complexity, as in the worst case they requires visiting all the list elements.

Guards

It may be convenient to add to the list some fake elements (called *guards*) which allow

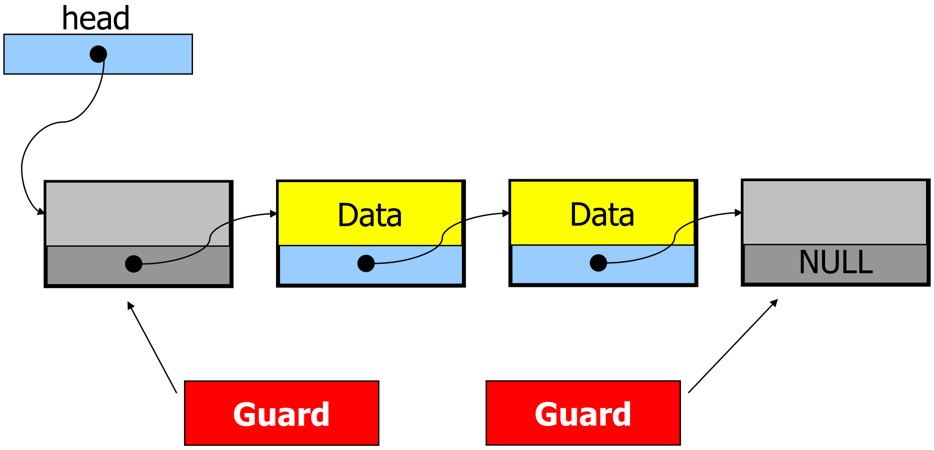
- Simplifying code for list management
- Improving efficiency, without changing its worst-case complexity.

Guards: unsorted lists

In this case 2 guards are used (one in the head, one in the tail).

Guards may simplify code as it is no more necessary to consider as separate cases insertion/deletion in head and tail.

Example



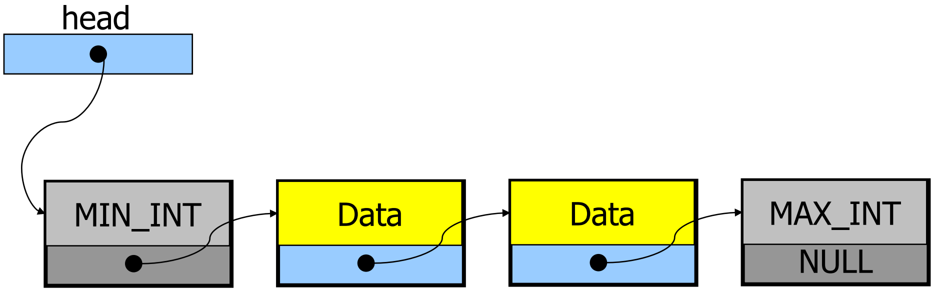
Guards: Ordered Lists

In this case guards contain the minimum and maximum value that can be stored.

In this way

- Code is more simple (insert/delete in head/tail are no more separated cases)
- Execution is faster , as test at the end of the list is now useless.

Example



Faster Algorithm

```
LIST_ELEMENT * search (LIST_ELEMENT *t, int val)
{  LIST_ELEMENT *p;
   p=t;
   while (p!=NULL)
   {  if (p->code==val)
         return  p;
      p=p->next;
   }
   return  p;
}
```

Each iteration contains two tests

➡

```
{   LIST_ELEMENT *p;
   p=t;
   while (p->code < val)
      p=p->next;
  if (p->code == val)
      return  p;
   else
      return NULL;
}
```

Each iteration contains only one test