

BM267 - Introduction to Data Structures

3. Elementary Data Structures

Ankara University
Computer Engineering Department

Objectives

Learn about **elementary data structures** - Data structures that form the basis of more complex data structures.

- **Structure:** Combines different data types as a unit.
- **Array:** Combines many instances of the same data type as a unit.
- **Linked list:** allows insertions and removals anywhere. Implementation using dynamic allocation vs fixed arrays.
- **String:** allows insertions and removals of substrings hence the size changes dynamically.

Self-referential objects

- **Object** = any data type (elementary or aggregate) that has memory allocated for it (we talk about concrete types now).
- **Node**: object that contains a pointer to a object of the same type
- Nodes can be linked together to form useful data structures such as lists, queues, stacks and trees
- By convention, a **NULL** pointer (value 0) is used to indicate that the link is not meaningful.

Node

- A node in the simplest form has two fields:

Data	Link
-------------	-------------

- A node can have two or more links, each pointing to a node.

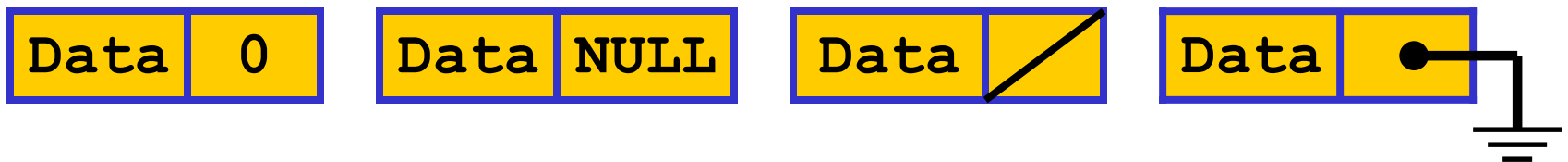
Data	Link	Link
-------------	-------------	-------------

- **Link** field always contains information that identifies the **successor** node.
- This information may be a pointer (memory address), or an index (into an array).

Node

Data	Link
------	------

- **Data** field can contain any data type.
- **Link** field points to a node (it contains a memory address)
- If link field has a value of **0** (**NULL** or **NIL**) then node does *not* have a successor.
- Then, “the link is terminated”:



Node

A node for int values defined in C:

```
struct node
{
    int data;
    node * next;
};
```



- You can define an array of such structures:

```
struct node array[N];
```

- Or allocate each node separately:

```
malloc(sizeof( struct node));
```

Programming errors with pointers

- Dereferencing a NULL pointer

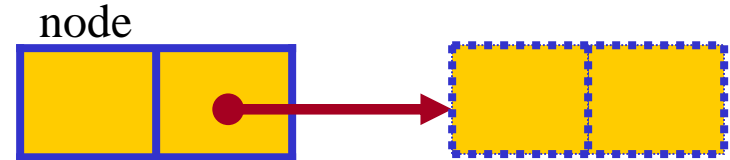
```
struct node *q = NULL;
```



```
q->data = 5;          /* ERROR */
```

- Using a freed element

```
free(q->next);
```



```
q->next->data = 6;    /* PROBLEM */
```

- Using a pointer before set

```
q = (node*)malloc(sizeof(node));
```

```
q->next->data = 7;    /* ERROR */
```

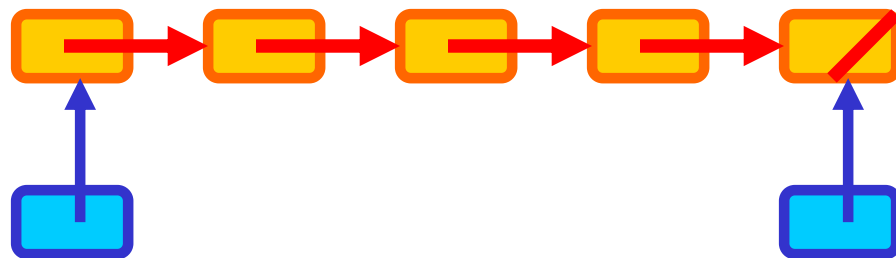
Linked List (as an elementary data type)

The simplest kind of **linked list** is a linear chain of links (pointers) to nodes of the same type.

More sophisticated linked lists may have several chains of pointers.

A linked list can only be reached thru its handle(s).

Handles are called the **head pointer** (or list-head), **tail-pointer** (or list-end).



Linked list - most common types

Singly linked list

- Each node points to the next
- Terminates with a null pointer
- Only traversed in one direction

Circular, singly linked

- Pointer in the last node points back to the first node

Doubly linked list

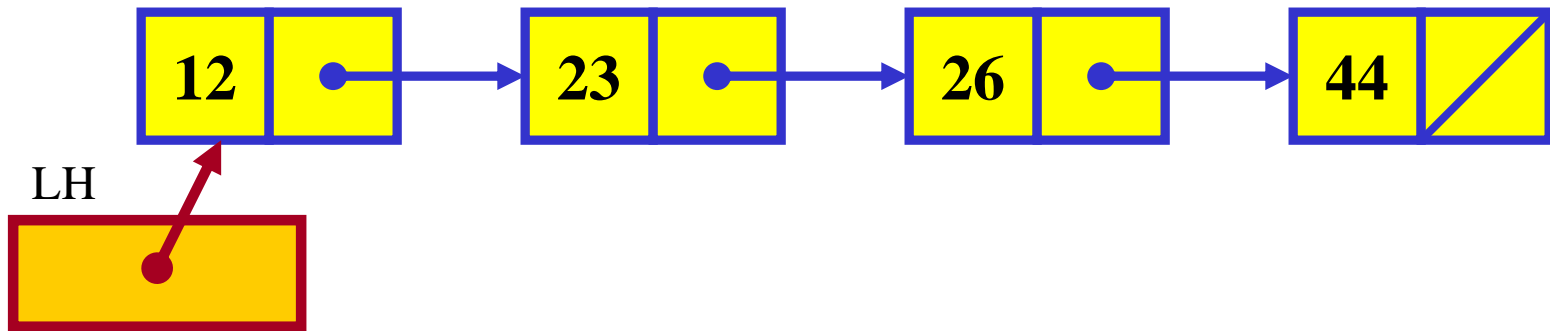
- May have two “start pointers” — head element and tail element
- Each node has forward / backward pointers
- Allows traversals both forwards and backwards

Circular, doubly linked list

- Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

Singly-linked list

- Consists of a sequence of nodes with one link field.



- The *first* node in the list is called the **head node**.
- To access a node, the **predecessor** of the node must be available.
- To access the head node, its address is kept in a special pointer named **head** (or **listhead**, **LH**) **pointer** outside the list.

Singly-linked list

List operations:

- **Insert** a new item (pointer to the new node is given)
- **Delete** an item (the key value of the item is given)
- **Search** for an item (the key value is given)

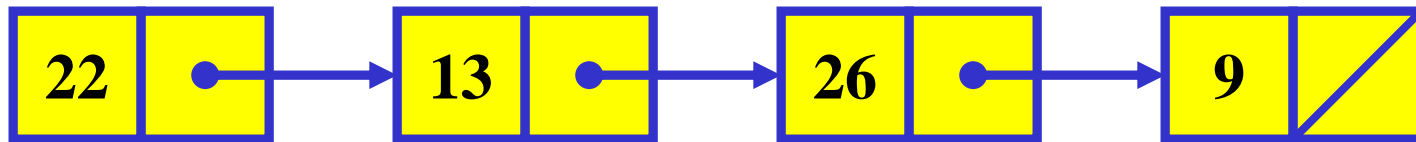
Normally, only the listhead (**LH**) is given.

There is no information on:

- The number of nodes (might be 0, 1 or any number N).
- The value in each node.

Singly-linked unordered list

Let's assume that the items are always added to the beginning of the list.



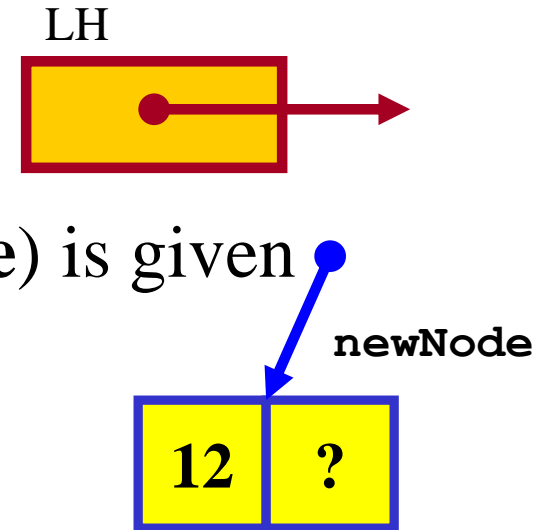
Deleting an item may require the traversal of the entire list.

Searching for an item may require the traversal of the entire list

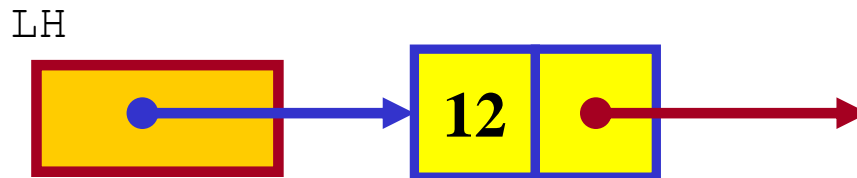
Singly-linked unordered list

Adding a node to an unordered list:

- List head pointer (LH) is given,
- A pointer to the new node (**newNode**) is given



The desired result is:



Singly-linked unordered list

```
struct LNODE
{
    int val;
    LNODE * next;
};
LNODE * LH;
LNODE * newNode;
```

C code for creating a node:

```
newNode = malloc(sizeof(LNODE)) ;
```

Singly-linked unordered list

C code for **adding a node** to the beginning of the list:

- Let the caller update the listhead pointer

Caller code: `LH = AddNode(LH, newNode);`

Where:

```
LNODE * AddNode(LNODE* head, LNODE* newNode)
{
    newNode->next = head;
    return newNode;
}
```

- Let the called function update the listhead pointer

Caller code: `AddNode(&LH, newNode);`

Where:

```
void AddNode(LNODE** head, LNODE* newNode)
{
    newNode->next = *head;
    *head = newNode;
}
```

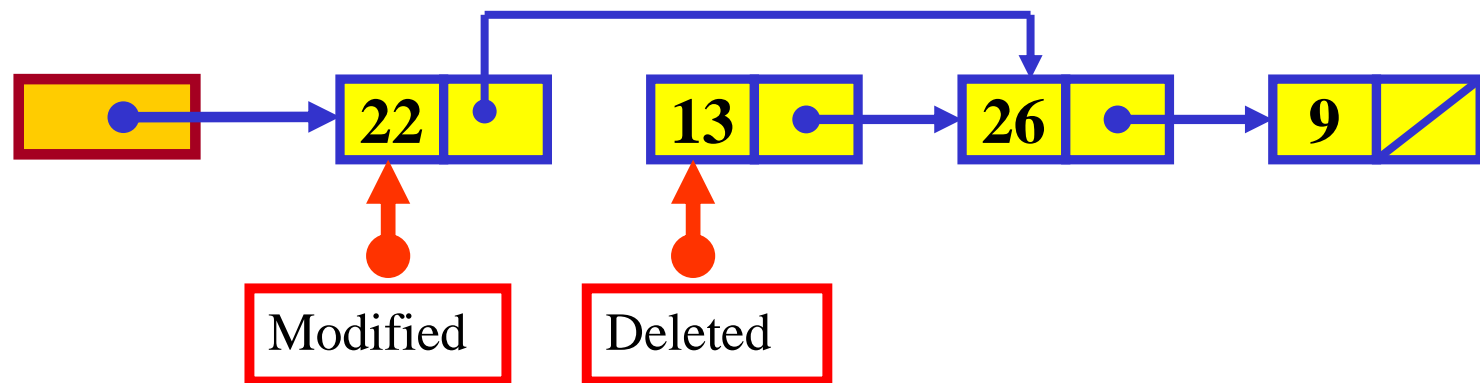
Singly-linked unordered list

Deleting a node:

- List head pointer (**LH**) is given,
- The key value of the item (**key**) is given (Assume 13)



The desired result is:



Singly-linked unordered list

For deletions, need to keep two pointers, pointing to the **modified** and **deleted** items.

Special cases: In case of deleting the **first** item, listhead pointer LH must be updated.

We will assume that an item with **key** always exists in the list.

Singly-linked unordered list

C code for **deleting a node** whose data value is given:

- Let the caller update the listhead pointer

Caller code: `LH = DeleteNode(LH, key);`

Where:

```
LNODE * DeleteNode(LNODE * head, int key)
{
    LNODE * node = head;
    LNODE * prev = NULL;
    while (node->val != key)
    {
        prev = node;
        node = node->next;
    }
    if (!prev)          (Deleting the first node?)
        head = node->next;
    else
        prev->next = node->next;
    free (node);
    return head;        (Return listhead)
}
```

Singly-linked unordered list

- Let the called function update the listhead pointer

Caller code: `DeleteNode (&LH, key) ;`

Where:

```
void DeleteNode(LNODE ** head, int key)
{
    LNODE * node = head;
    LNODE * prev = NULL;
    while (node->val != key) (Find node to delete)
    {
        prev = node;
        node = node->next;
    }
    if (!prev) (Deleting the first node?)
        *head = node->next;
    else
        prev->next = node->next;
    free (node);
}
```

Singly-linked unordered list

Search for a value:

- List head pointer (**LH**) is given,
- The key value (**key**) is given



The desired result is:

- TRUE : Value is in the list
- FALSE: Value is not in the list

Singly-linked unordered list

Caller code:

```
if (Search (LH, 12) ) /*Search for an item with value 12 */  
    ... /* Value found */  
else  
    ... /* Value not found */
```

Where:

```
int Search(LNODE * node, int key)  
{  
    while (node)  
        if (node->val == key)  
            return 1; /* Value found */  
        else  
            node = node->next;  
    return 0;          /* Value not found */  
}
```

Sorted lists

Keep the items on the list in a sorted order, based on data value in each node



Advantages:

- already sorted, no need for sort operation
- operations such as delete, find, etc. need not search to the end of the list if the item is not in list

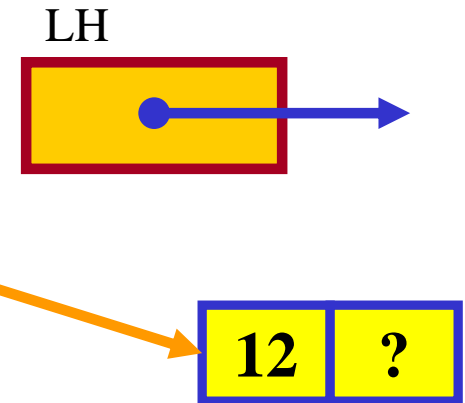
Disadvantages

- Insert operation must search for the right place to add element (slower than simply adding at beginning)

Singly-linked ordered list

Adding a node to an ordered list:

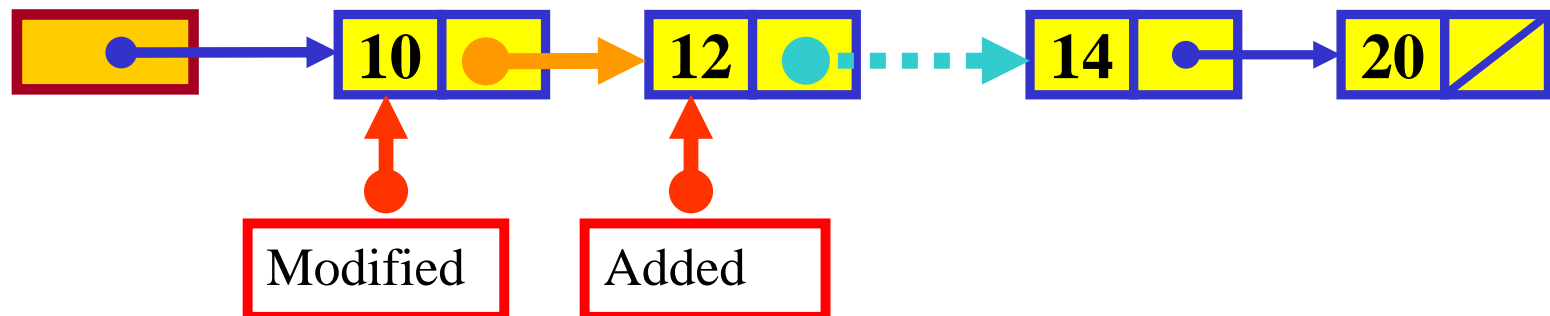
- List head pointer (LH) is given,
- A pointer to the new node is given



Initial configuration

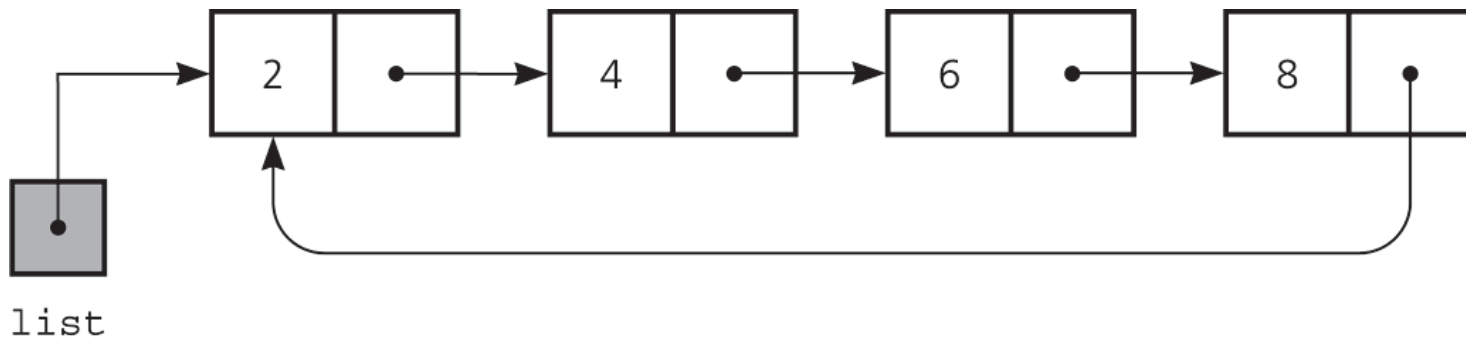


The desired result is:



Circular singly-linked list

- Last node references the first node
- Every node has a successor
- No node in a circular linked list contains **NULL**

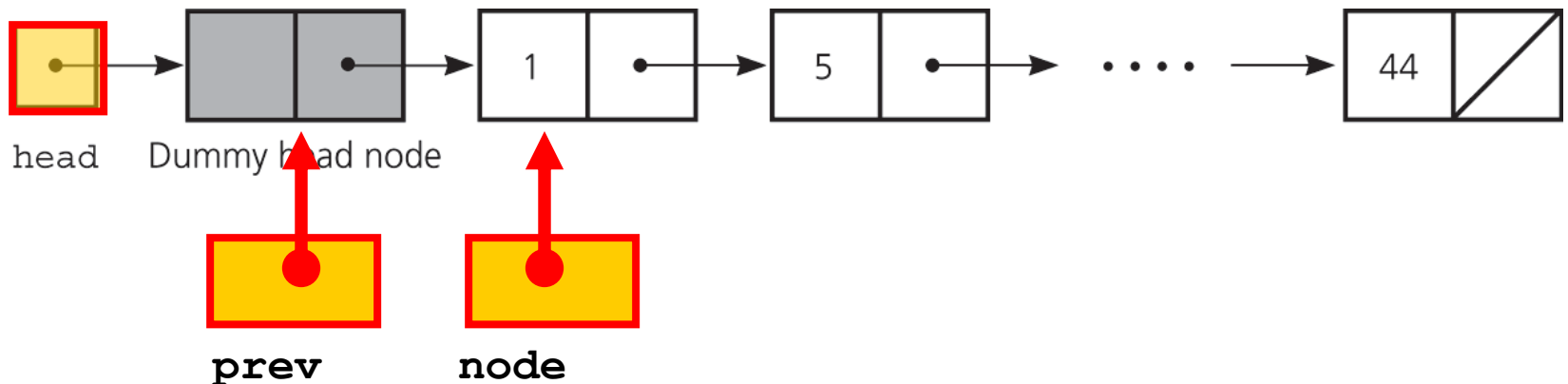


Checking if the circle is completed:

if (node->next == list) ... Pointer comparison

Singly-linked list with dummy head node

- Dummy head node is always present, even when the linked list is empty.
- Insertion and deletion algorithms use two pointers, **prev** and **node**,
- For empty lists, initialize **prev** to reference the dummy head node, rather than **NULL**.
- Move both pointers together.



Doubly-linked list

A double link node for int values defined in C:

```
struct LNODE
{
    LNODE * llink;
    int val;
    LNODE * rlink;
};
```

llink	data	rlink
-------	------	-------

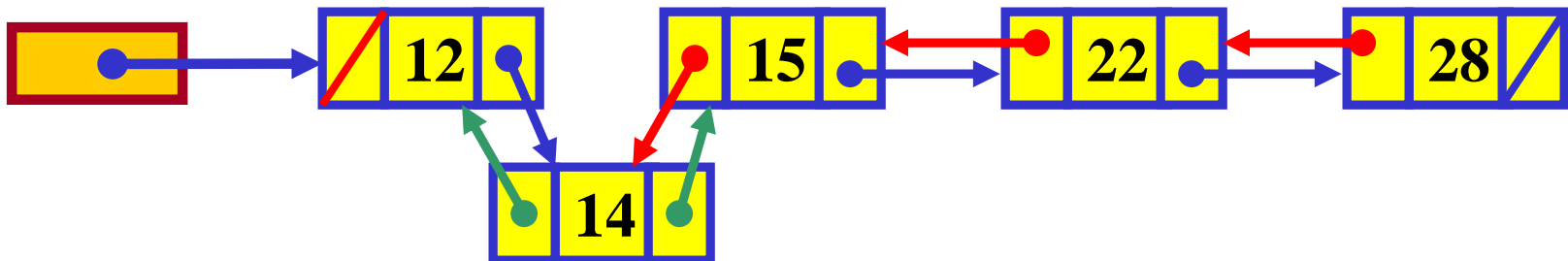
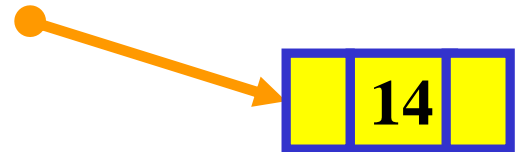
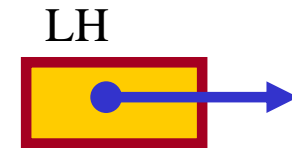
prev	data	next
------	------	------



Doubly-linked ordered list

Adding a node to an ordered list:

- List head pointer (LH) is given,
- A pointer to the new node is given



Doubly-linked ordered list

newNode



Adding a node to an ordered list

(Assuming LH is global)

Node = LH

(Initialize node pointer)

WHILE (node \neq NULL)

{

IF (node->val > val)

(Find the location to insert)

{ newnode->rlink \leftarrow node

newnode->llink \leftarrow node->llink

node->llink \leftarrow newnode

IF (newnode->llink == NULL)

(Adding as first?)

LH = newnode

(Update listhead)

ELSE

(newnode->llink)->rlink \leftarrow newnode

break

}

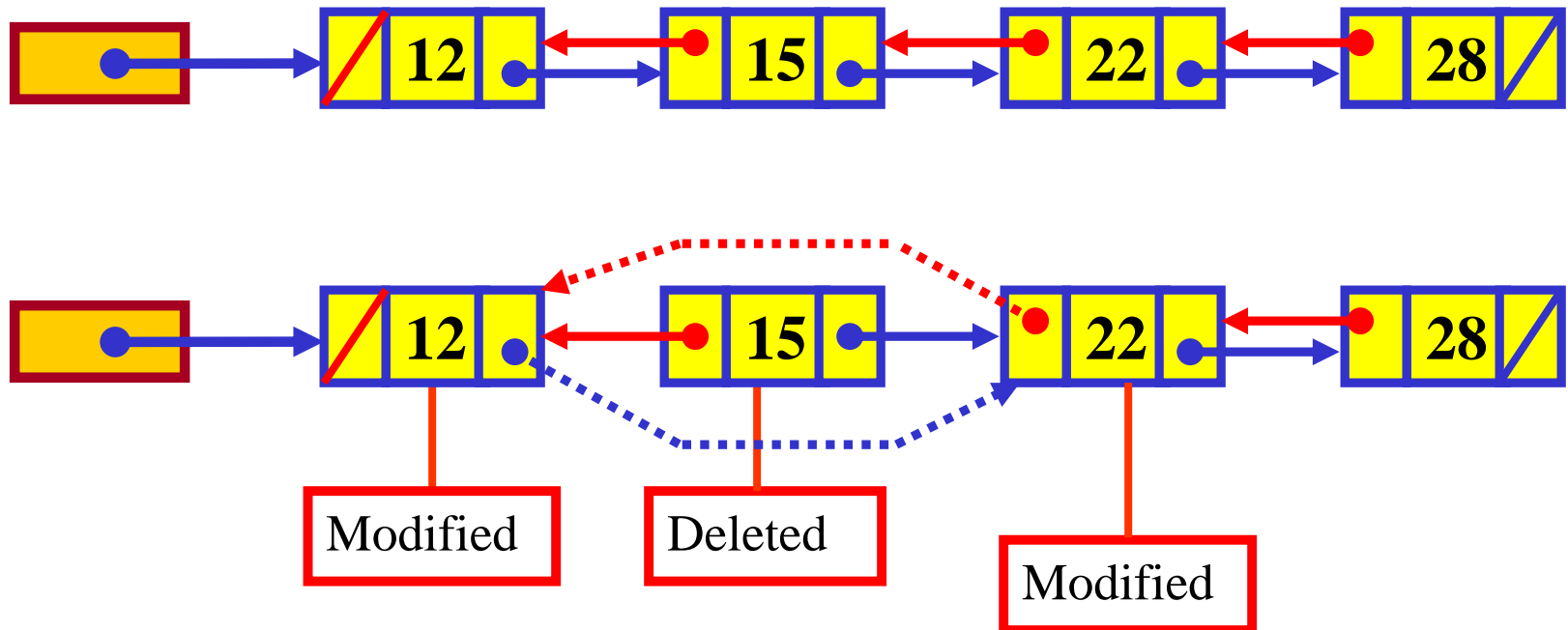
ELSE node \leftarrow node->rlink; (keep trying)

}

Doubly-linked ordered list

Deleting a node:

- List head pointer (**LH**) is given,
- The key value of the item (**val**) is given (Assume 15)



Implementing lists using arrays

- Arrays have fixed number of elements (check for overflow).
- Pointers now become array indices (of type integer).
- Since C arrays start with index 0, we will assume **-1** corresponds to the **NULL** pointer.
- The last used array element need to be maintained.
- Here is the **NODE** structure for use with arrays in C:

```
struct NODE
{
    int val;
    int next;
};
```

Implementing lists using arrays (2)

```
NODE node[N];  
int LH = -1;
```

A NODE array of size N,
(N is a compile-time constant)

Note that we have to distinguish **unused** array elements. We will accept the convention that a link value of -2 denotes an unused array element.

Therefore, the array has to be initialized.

```
for(i=0; i<N; i++) node[i].next = -2;
```

- 1 means: 'the chain ends here',
 - 2 means: 'node is available (not currently used).'
- Any other value means: 1) 'node is currently in the list' and,
2) 'is followed by node whose index is here'.

-1

LH

	-2
	-2
	-2
	-2

Implementing lists using arrays (3)

We will also need to find an **empty element** to insert the new node (instead of using malloc()).

```
Caller Code:  index = FindEmpty( );
               if (index == -1)
               { /* No more space on array */}
               else
               { /* Use node[index] */ }
               ...
```

```
Where:        int FindEmpty()
               { int i;
                 for(i=0; i<N; i++)
                   if(node[i].link == -2)
                     return i;
                 return -1;
               }
```


Implementing lists using arrays (4)

C code for **adding a node** to the beginning of the list, assuming **node []** and **LH** are defined globally.

Caller code:

```
AddNode (newNode) ;
```

Where:

```
void AddNode(int newVal)
{  int index;
   index = FindEmpty( );
   if (index == -1)
   { /* No more space on array */ }
   else
   { node[index].val  = newVal;
     node[index].next = LH;
     LH = index;
   }
}
```

Implementing lists using arrays (5)

C code for **deleting a node**, assuming **node []** and **LH** are defined globally and **val** always exists in the list.

Caller code: `DeleteNode(val);`

Where:

```
void DeleteNode(int delVal)
{ int index=LH;
  while(! (index < 0))
    if(node[index].val == delVal)
      break;
    else
    { prev = index;
      index = node[index].next;
    }
  if(prev < 0) (Deleting the first node?)
    LH = node[index].next;
  else
    node[prev].next = node[index].next;
  node[index].next = -2; (Mark as empty)
}
```

Implementing lists using arrays (6)

C code for searching a value in an unordered list, assuming **node []** and **LH** are defined globally.

```
Caller code:  if (SearchVal (val) ) /*Search for an item with value val */  
              ... /* Value found */  
            else  
              ... /* Value not found */
```

```
Where:        int SearchVal(int val)  
              { int index=LH;  
                while (! (index < 0))  
                  if (node[index].val == val)  
                    return 1;    /* Success */  
                  else /* keep trying */  
                    index = node[index].next;  
                return 0; /* Failure*/  
              }
```

Lists vs. Arrays - A comparison

Space (storage) considerations

- A linked list requires pointers to nodes.
- An array requires the maximum number of elements to be known in advance. If that maximum is not required, space is wasted at the end of the array.

Time considerations

- Operations on a linked list require more lines of explicit code than those in an array. However, addressing an array element uses more implicit (compiler generated) code.
- Arrays are quicker at finding and altering ‘in the middle’
- Linked lists are quicker at insertions and removals ‘in the beginning/middle’

String (as an elementary data type)

- Not a built-in C/C++ type.
- String type can be (and is) implemented transparently.
- Need to grow and shrink in size - Efficiently represented as (variable-length) array of characters.
- Need to be maintained dynamically (by the system or by the user.)
- String Operations:
 - Compute length
 - Copy
 - Compare strings
 - Check substring existence
 - Append, insert, delete substring

String representation

Allocate a fixed number of bytes for characters.

Use as many characters as needed, disregard the rest.

```
char str1[50];
```

A	N	K	A	R	A		U	N	I	V	E	R	S	I	T	Y	?	?	?	?	?
---	---	---	---	---	---	--	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Array elements can be addressed individually as **chars**.

Beginning of the string is the first character: **str1[0]**.

Handle of the string is a pointer to the first character.

How to tell at which position the string ends?

String representation

Two accepted ways to represent strings.

- **ASCIIZ representation:** End the string with a binary 0.

A	N	K	A	R	A		U	N	I	V	E	R	S	I	T	Y	\0	?	?	?	?
---	---	---	---	---	---	--	---	---	---	---	---	---	---	---	---	---	----	---	---	---	---

Null (empty) string:

\0	?	?	?	?	?	?	?	?	?	?	?	?	?
----	---	---	---	---	---	---	---	---	---	---	---	---	---

- **Size-Content representation:** Keep string size at the front.

17	A	N	K	A	R	A		U	N	I	V	E	R	S	I	T	Y	?	?	?	?
----	---	---	---	---	---	---	--	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Null (empty) string:

00	?	?	?	?	?	?	?	?	?	?	?	?	?
----	---	---	---	---	---	---	---	---	---	---	---	---	---

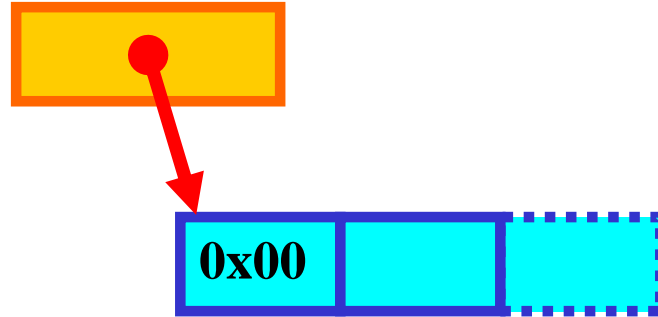
Null string representation

`char * str;` An empty (null) string is **NOT** represented by:
`str = 0;`

00000000

An empty (null) string is represented by:

`*str = 0;`



- Empty or not, string pointers must always point to valid memory locations.
- If a string pointer is **NULL**, the string is invalid (cannot do string operations on it)
- A **‘null pointer’** and a **‘null string’** are different entities.

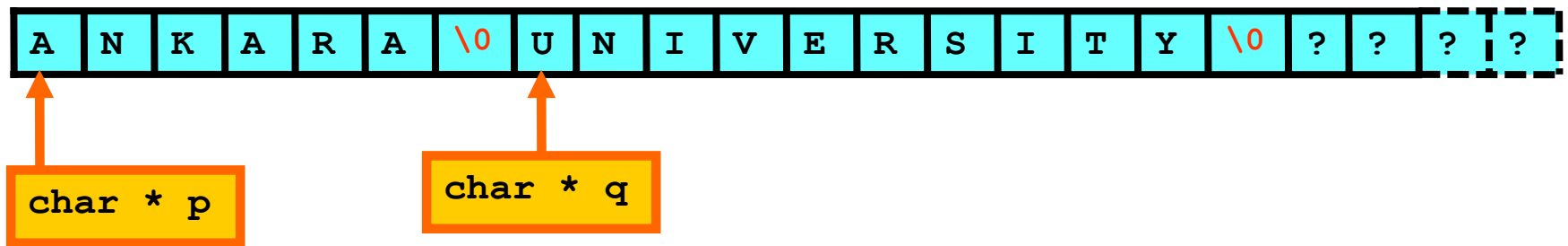
String buffer

- When processing strings, strings are allowed to grow/shrink in size (create, copy, append).
- Allocating one (maximum size) array for each string is very inefficient.
- Compilers use a contiguous memory block called **string buffer** (or **string space**) to keep literal strings which are constant (format strings, initialization strings etc.)
- Some applications define a **string buffer** which is a memory area that contains all strings side by side, and can be manipulated under program control.

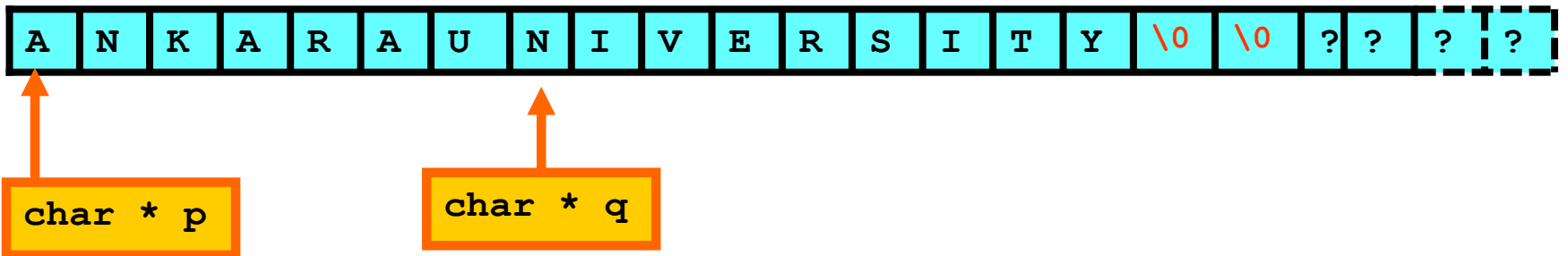
A	N	K	A	R	A	\0	U	N	I	V	E	R	S	I	T	Y	\0	?	?	?	?
---	---	---	---	---	---	----	---	---	---	---	---	---	---	---	---	---	----	---	---	---	---

String buffer

If a string grows in size, it may need to be moved to a different location in string buffer.

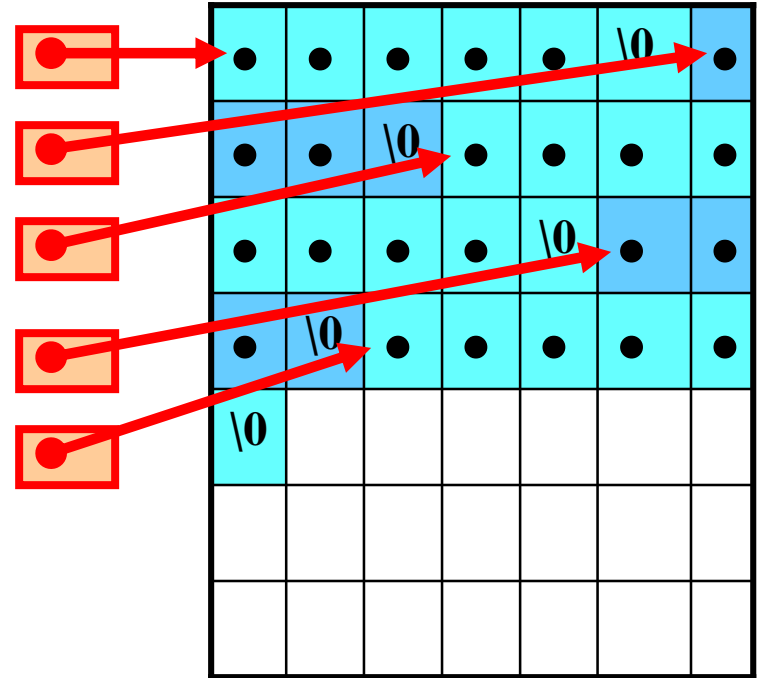
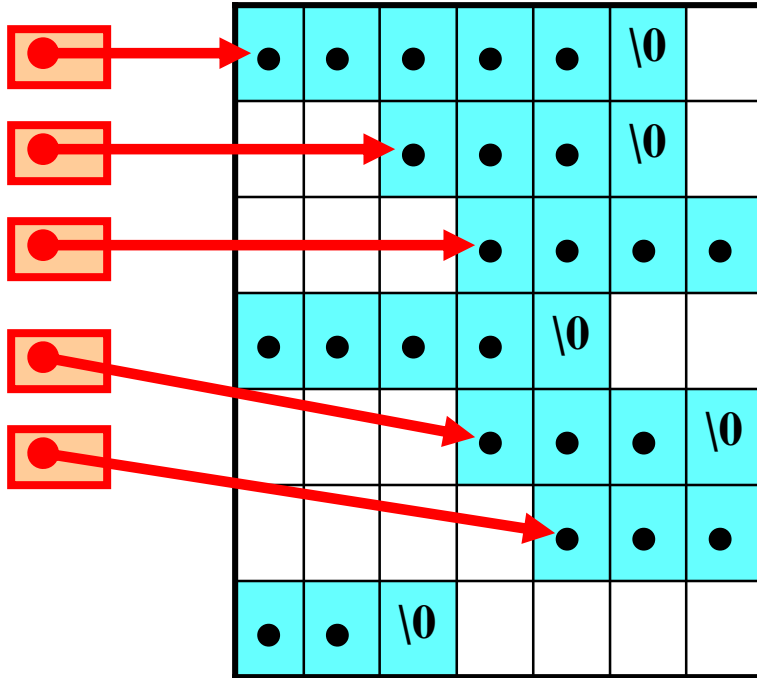


After `strcat(p,q)` :



String buffer

Empty locations in string buffer may need to be compacted from time to time.



Elementary data structures

Elementary data structures are the data types that are implemented in programming language syntax, or can be added with little effort.

Basically, structures, arrays, linked lists and strings are sufficient to implement most of the useful data structures.

Many languages have elementary types as syntactic (language-defined) data types, or have extensive libraries that implement them.