REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

Université de M'sila
Faculté des Mathématiques et de l'Informatique
Département d'informatique

جامعة المسيلة
كلية الرياضيات والإعلام الآلي
قسم الإعلام الآلي

Chapter 3: Part 2

Pointers and linked lists

Algorithmics and data structure 2

Presented by : Dr. Benazi Makhlouf
Academic year : 2023/2024

Content of chapter 03 part 2:

5. Linked lists
6. Operations on linked lists
7. Doubly linked list
8. Special linked lists
    1. Queues
    2. Stacks

1. Introduction

Arrays represent an important concept in any programming language because they allow quick access to their elements. However, they have two disadvantages:

1. Array elements must be contiguous (adjacent) in memory.
2. It is not possible to insert or delete elements in the array without recreating the array.

Therefore, we need another data structure known as a **linked list.**

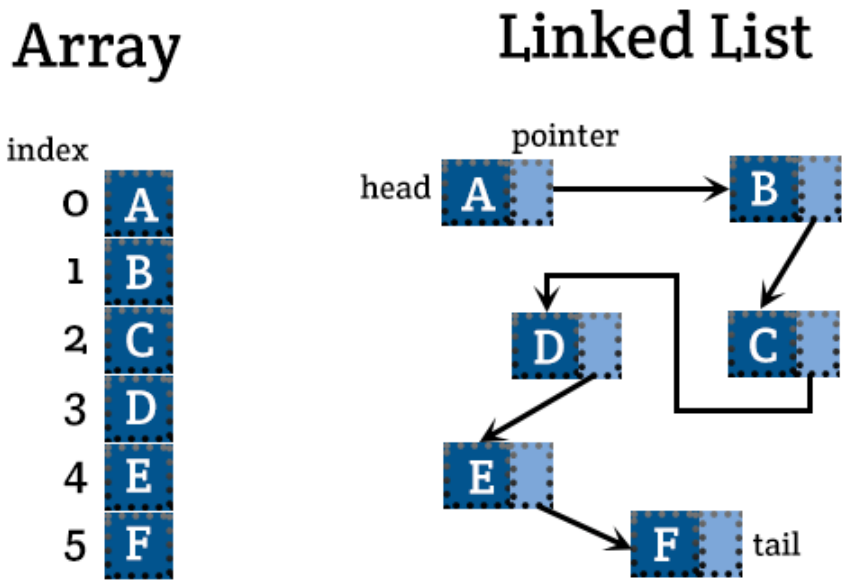
Definition

- Lists (linked lists) are a recursive data structure. They are composed of dynamically created nodes of the same type, linked to each other by pointers. Unlike arrays, nodes in linked lists can be in noncontiguous locations in memory. Linked lists are made up of elements called records, elements, nodes, or cells. Each item contains:

  - One or more fields to store data.
  - A pointer (link) to the next element in the list.

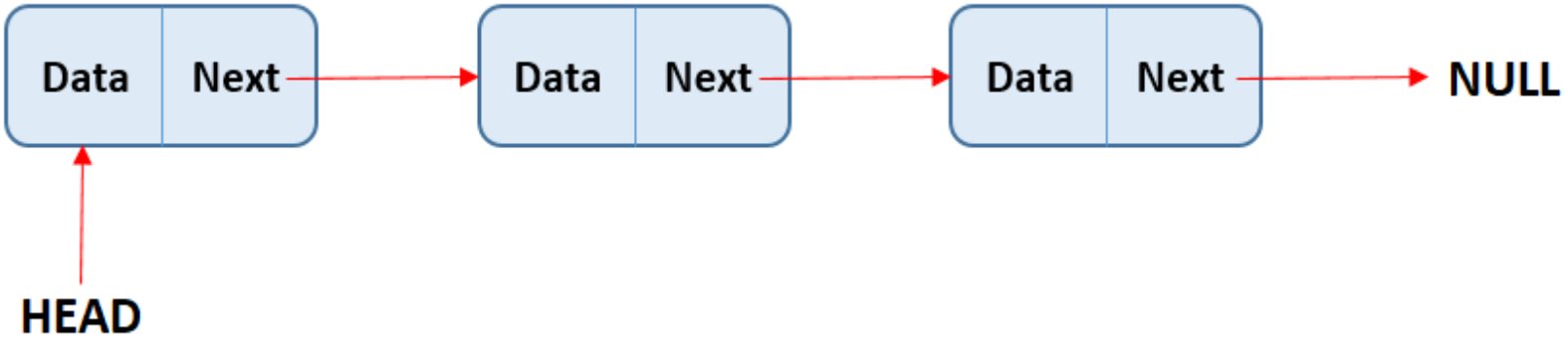
Linked Lists vs Arrays

- Linked lists allow dynamic size changes through the insertion or removal of elements from any position in the list.
- To access an element in a linked list, you must traverse through all the preceding elements, starting from its header,. This process may take longer compared to the direct access of elements in an array.
- Linked lists are considered a linear data structure, in contrast to arrays that allow for random access.



The representation

- In the C language, a node in a linked list is typically represented using a 'struct' structure, while a header is represented by a pointer.
- To simplify the explanation, a single data field of integer type, often named 'data', is used for all records in the list. This is done instead of using specific data fields for each record type, such as information about students (e.g., name, first name, date), making the explanation more simple .



Declaration

Declaration of the type of elements or nodes

```
typedef struct Node{
  int data ;
  struct Node * next;
} Node;
```

```
structure Node
  data : integer
  next: ^Node
end_structure
```

- **`data`** represents the data stored in the list, such as a student's first and last name, the date of an event, etc. This field can be replaced by any other variable corresponding to the type of data you want to store in the list.

- **`next`** is a pointer that contains the address of the next element in the list, or **NULL** if it does not point to any element. This field is important because it allows the nodes to be linked to each other to form the linked list.

head type declaration

```
typedef Node* List;
```

First element

```
List h ; // a simple pointer to a structure
Node e1,e2; // a structure
h =&e1;
e1.data=1; ⇔ (*h).data=1;
e1.next=&e1; ⇔ (*h).next= &e1;
e2.next= NULL; ⇔ (*(*h).next).next = NULL;
```

We can create a dynamic Node element as follows:

```
 h =(Node*) malloc (sizeof Node ) ;
```

Or in C++

```
 h = new Node;
```

The selection operator -> in C

The selection operator '->' in C is used to access the fields of a structure through a pointer. The expression `'h = &e1;'` means that the pointer 'h' points to the memory address of the object 'e1'. To access the fields of this object, such as 'e1.data' or 'e1.next', we can use the syntax `'(*h).data'` and '(*h).next'. However, in the C language, it is more common to use the field selection operator '->', resulting in `'h->data'` and `'h->next'`. This avoids the use of the dereference operator ‘(*),' making the code more concise and readable.

Example:

```
h->data = 1;  // Equivalent to (*h).data = 1;
h->next = NULL;
```

Comments:

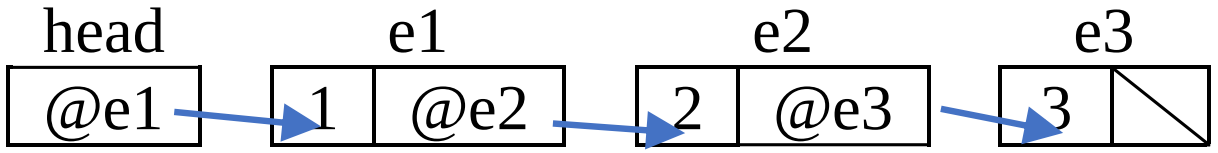
- `'h.data'` is incorrect because 'h' is a pointer, not a structure.
- Both 'h' and 'e1.next' pointers are of the same type, allowing assignments between them.
- '->' has a priority over '*’”

Traversing a linked list:

The last element: The last element in the list has no next element, so **NULL** is assigned to its 'next' pointer.

When traversing the list, **NULL** is used to check whether the last element has been reached or not.

The traversal: The following example demonstrates how to traverse the elements of a linked list. Suppose we have the following list:



As 'e1.next' points to 'e2', we can make 'h' point to 'e2' by performing the following operation: 'h = e1.next;'.

Now that 'h' points to 'e2', then 'h->next' is equivalent to 'e2.next'.

`h= &e2` ⇔ `h= e1.next` ⇔ `h= h->next`

Therefore, to move from one node to another, we use 'h = h->next'.

Traversing

```
while (h != NULL){
   //do something
   h = h->next;
}
```

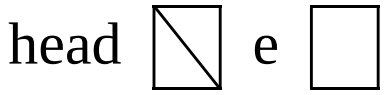
```
while ( h≠NULL ) do
   //do something
   h←h^.next
End while
```

```
while (h){
   //do something
   h = h->next;
}
```

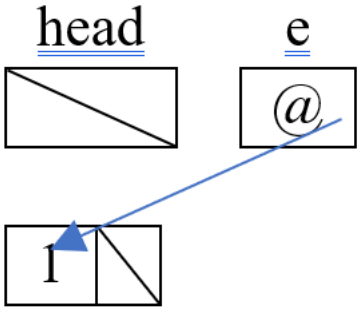
Creation

Suppose we have an empty list with ‘`h =NULL;`’ To create a new element, we use the dynamic memory allocation function malloc ().

```
List e, h = NULL;
```

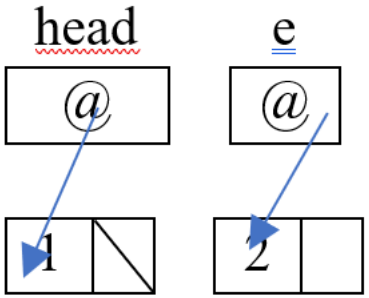



```
e = new Node;
e ->data=1;
e ->next= NULL;
```




```
h = e;
```

```
e = new Node;
e->data=2;
```



element “e” has been created and can be added at the top of the list by linking it to the first element in the list using its next field.

```
e->next=h; h=e;
```

or at the end

```
h->next=e;
```

6. Operations on linked lists

There are several ways to create functions to add or remove an item from a list:

1. Using functions that take a list as a parameter and return a list. In this case, the list can be passed by value.

2. By using procedures and an auxiliary element (sentinel) to avoid passing by address. In this case, the list can be passed by value.

3. Using procedures without auxiliaries. In this case, the list must be passed by address.

4. By using functions that take a list as a parameter and return a Boolean value (bool) to indicate whether the operation was successful (true) or not (false). In this case, the list must be passed by address.

We will use the latter method.

6.1. Displaying a list

```
  void display_list (List h) {
    while (h != NULL) {
      printf("%d->", h->data);
      h = h->next;
    }
    printf("end\n");
  }

void display_list (List h) {
  if (h){
    printf("%d->", h->data);
    display_list (h->next);
  } else printf("end\n");
}
```

6.2. List size

```
int size_list(List h){
  int n=0;
  while (h != NULL) {
    h= h->next;
    n++;
  }
  return n;
}
```

```
int size_list (List h){
  if (!h) return 0;
  return 1+ size_list (h->next);
}
```

6.3. Add an item to the list

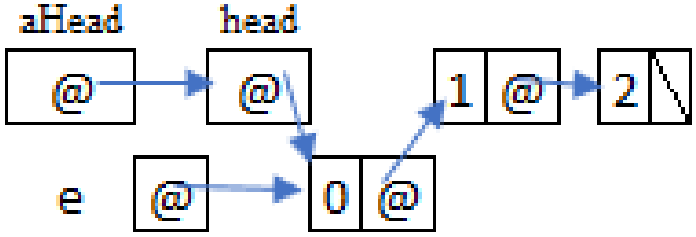
The process of adding an element to a linked list occurs in 3 steps:

1. Create and initialize a node
2. Determine the location of the node.
3. Add the node to the list by reassigning pointers.

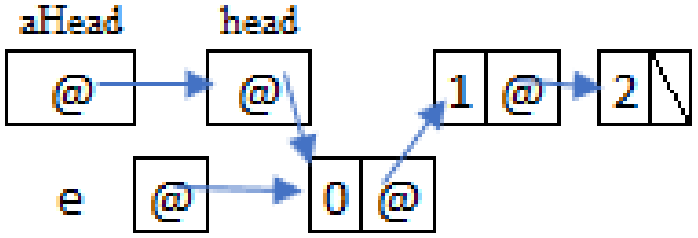
Add an item to the beginning of the list (at the head)

Adding an element to the beginning of the list requires changing the 'h' (head) of the list. Therefore, it is important to pass the list by address so that this change is visible to the caller.

1. Create a new item, and if the creation fails, return false to the caller.
2. Initialize the new element.
3. Replace 'next' of 'e' to point to the first element in the list.
4. Change the 'h' (head) of the list to point to the new element.
5. Note: 'aHead' and 'e' are two local variables that are removed immediately after the procedure is executed.



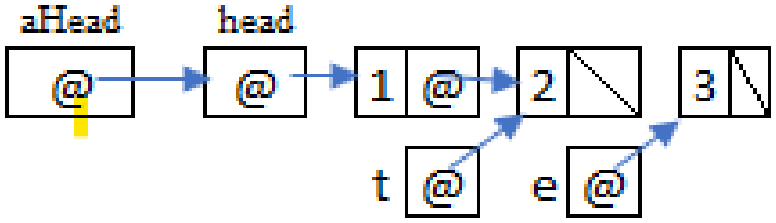
```
int add_head (List &h , int d){

 List e = new Node;
  if (e == NULL)
    return 0 ;


  e->data = d;
  e->next =h;

   h =e;
   return 1;
 }
```



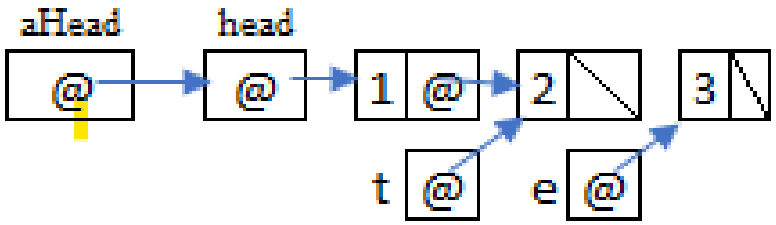
Add an element at the end

To add an element at the end, we may need to modify the list header, so it is essential to pass it by address. Here are the steps:

1. Create a new item.
2. Initialize the element and assign **NULL** to "next" because it will be the last element.
3. If the list is empty, insert the element into the header 'h.'
4. If the list contains at least one element, search for the last element.
5. Insert the element last.



```
int append_end (List &h, int d){
  List t;
  List e = new Node;
  if (e == NULL) {
    return 0 ;
  }
   e->data = d;
   e->next = NULL;
  if (h == NULL)
    h = e;

 else {
   t= h ;
   while (t->next != NULL)
     t= t->next;

   t->next=e;
  }
  return 1;}
```



6.4. Remove an item from the list

The process of removing a node from a list involves four steps:

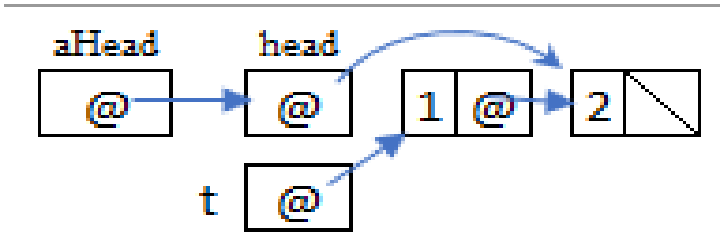
1. Determine the previous node of the node to be deleted.
2. Keep the address of the node to be deleted in a variable.
3. Connect the previous node to the next node of the node to be deleted.
4. Empty the memory reserved by the node to be deleted.

There are three possible cases: either the list is empty, contains a single element, or contains more than one element.

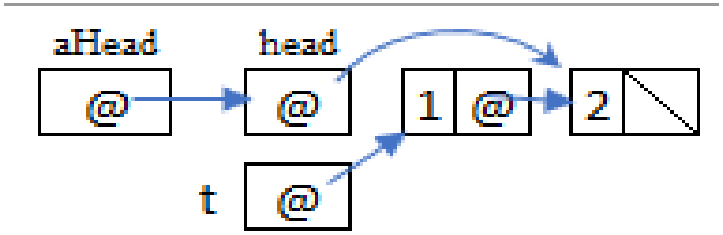
Remove element from head

We may remove an element from the header, so it is necessary to pass the list by address. Here are the steps:

1. If the list is empty, there is no element to delete, so false is returned.
2. Store the address of the first element to delete in 't.'
3. Connect 'h' with the second element.
4. Clear the memory reserved by the first element.



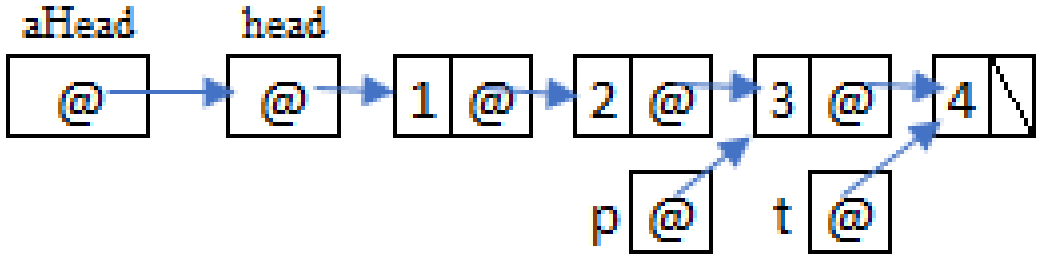
```
int delete_h (List &h ) {
   List t;


   if (h == NULL)
      return 0 ;

   t = h ;

   h =t->next;

  delete t;
  return 1;
}
```



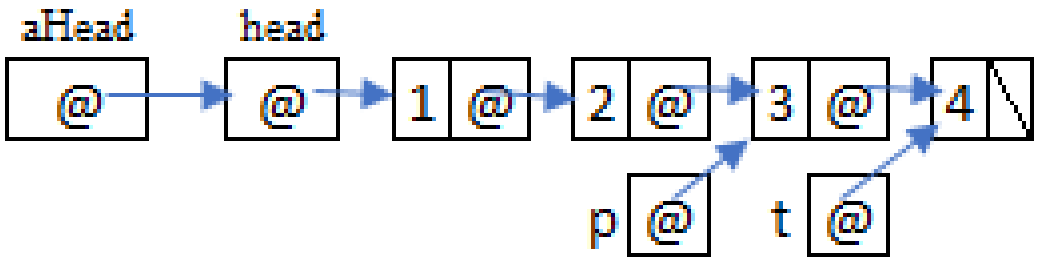
Remove an element from the end

We may remove an element from the head 'h,' so it is necessary to pass the list by address. Here are the steps:

1. If the list is empty, there is no element to delete, so we return false.
2. If the list contains only one element, delete it directly from the head 'h.'
3. If the list contains more than one element, we look for the last element 't' and the penultimate 'p.'
4. We assign **NULL** to the 'next' of the penultimate 'p' because it has become the last, and we delete the last element 't' from memory.



```
int delete_end (List &h ) {
  List t, p;

  if (h == NULL)
     return 0 ;

  if ((h )->next ==NULL) {
    delete h;
    h = NULL;
    return 1;
  }
  t = h ;
  while (t->next != NULL) {
    p=t;
    t= t->next;
  }

  p->next=NULL;
  delete t;
  return 1;
}
```



6.5. Deleting a list

1. We remove from the header until the list becomes empty
2. Or using delete_h function until it returns false

```
void delete_list (List &h ) {
  List t;
  while (h != NULL) {
    t = h ;
    h =t->next;
    delete t;
  }
}
```

```
void delete_list (List &h ) {
  while ( delete_h (h));
}
```

6.6. Main program (use)

```
int main() {
  List h =NULL;
  add_head (h, 3);
  add_head (h, 2);
  append_end(h, 4);
  add_head(h, 1);
  append_end(h, 5);
  printf("size=%d\n", size_list (h));
  display_list (h);
  delete_h(h);
  delete_end(h);
  printf("size=%d\n", size_list (h));
  display_list (h);
  delete_list (h);
  printf("size=%d\n", size_list (h));
  display_list ( h );
  return 0;
}
```

The screen

- The program will display
  - size=5
  - 1->2->3->4->5->end
- Then it will display
  - size=3
  - 2->3->4->end
- At the end it will display
  - size=0
  - end

End of part 2 of Chapter 03