5. Linked Lists

5.1. Introduction

In programming, to process data of the same type (e.g., student information), we need Arrays. Arrays are an important concept in any programming language because they allow quick access to their elements. However, they have two drawbacks:

- The elements in the array must be contiguous in memory.
- It is not possible to insert or delete items in the table without recreating the table again.

So we need another data structure known as a **Linked list**.

5.2. Definition

Linked lists are a recursive data structure composed of nodes of the same type, connected to each other by pointers. Unlike arrays, these nodes can be in non-contiguous locations in memory. Linked lists are made up of items (records, nodes, or cells), and each item contains one or more fields to store data and a pointer (link) to the next item in the list.

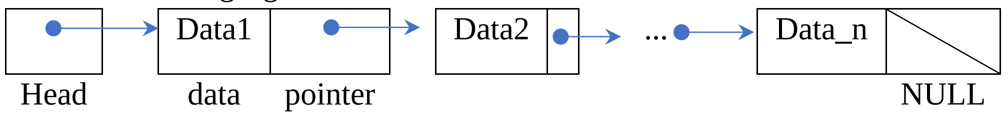
This structure allows you to change its dimension by inserting or removing items from any position in the list. To access any item in the list, you have to start from its header and go through all the items before it, which can take longer than going directly to the items in an array. So we say that it is a linear data structure as opposed to the array structure that allows random access.

5.3. Representation

In C, a node is represented using "struct" structures, while a header is represented by a pointer.

To simplify the explanation, we use a single integer data field called "data" for all records in the list, instead of using specific data fields for each record type (such as student information, last name, first name, date, etc.).

The following figure illustrates the structure of linked lists:



5.4. The Declaration

Declaring the type of elements or nodes

C	Algorithm

`typedef struct Node {`
`  int data;`
`  struct Node * next;`
`} Node;`
	`Node structure`
	`  Data : integer`
	`  next :^ Node`
	`end_structure`

In the structure of linked lists, "data" represents data stored in the list, such as a student's first and last name, the date of an event, and so on. This field can be replaced by any other variable that corresponds to the type of data you want to store in the list.

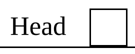
The "next" field is a pointer that contains the address of the next item in the list, or NULL if it doesn't point to any item. This field is important because it allows the nodes to be linked to each other to form the linked list.

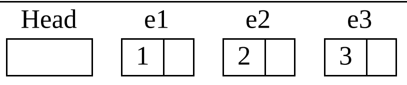
Declaring the Header Type

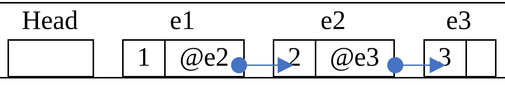
typedef Node* List;	**Type** List: ^ Node

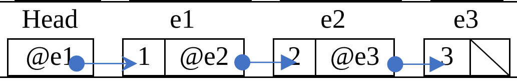
This means that List is the same as Node*.

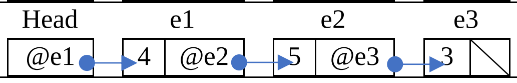
Example

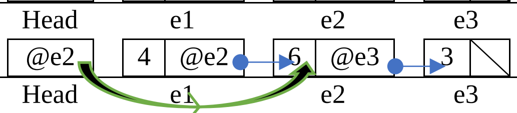
`List head;`	**var** head: List	A simple pointer-type variable that points to the first element 
`Node e1, e2,e3;`	e1, e2, e3:Node	3 compound variables of `type Node`

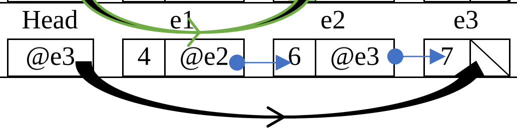
`e1.data=1;`
`e2.data=2;`
`e3.data=3;`
	e1.data←1 e2.data←2
	e3.data←3
		

`e1.next=&e2;`
`e2.next=&e3;`
	e1.next←@e2 e2.next←@e3	

`e3.next= NULL;`
`head=&e1;`
	e3.next← NULL
	head←@e1
		

`head->data=4;`
`head->next->data=5;`
	head^.data←4
	(head^. next)^.data←5
		

`head= head->next;`
`head->data=6;`
	head← head^.next;
	head^.data←6;
		

`head= head->next;`
`head->data=7;`
	head← head^.next;
	head^.data←7;
		

The -> operation in C language

Since the 'head' pointer points to the 'e1' element, the variable pointed to by 'head' and 'e1' are equivalent, so the expression '(*head).next' can be used to access the 'next' field of the 'head' point element instead of 'e1.next'

In C, we use the '->' operator instead of '(*).' to access the fields of the structure pointed to by 'head'. The expression 'head->next' is therefore equivalent to '(*head).next' to access the 'next' field of the element pointed to by 'head'

`e1.next` ⇔ (*head). `next` ⇔ head->next

`e1.data=5;` ⇔ `(*head).data=5;` ⇔ `head->data=5;`

Note that head.data is incorrect because head is a pointer, not a structure.

The 'head', 'e1.next', 'e2.next' and 'e3.next' pointers are all of the same type, which means that it is possible to make assignments between them.

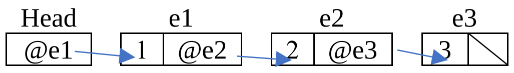
The Last Element

The last item in the list has no next item, so its 'next' pointer is assigned to NULL. When traversing the list, NULL is used to check whether the last item has been reached or not.

```
e3->next= NULL;
head= NULL; It's an empty list
```

The traversing of a linked list

The following example shows how to navigate through items in a linked list.
Let's say we have the following list:



Since "e1.next" points to "e2", we can make "head" point to "e2" by doing the following: "head = e1.next;".

Now that "head" points to "e2", then "head->next" is equivalent to "e2.next".

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

So, to switch from one node to another, we use "head=head->next"

while (head**!=** NULL){
   do something
head = head**->next**;
}
	while (head≠NULL) **do**
	   Do a southings
	   head←head**^**.Next
	end while

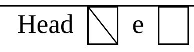
To access all the items in the list, we repeat the process until head takes the value of next from the last node, which is NULL.

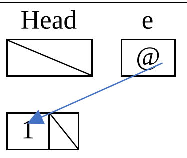
Observation: while (head!= NULL) ⇔ while **(head**)

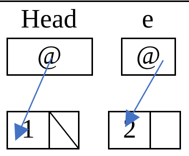
5.5. The Creation

To create a linked list, memory is dynamically reserved from a simple pointer variable.

Suppose we have an empty list with head=NULL; To create a new item, we use the malloc() dynamic memory allocation function.

List e, head= NULL;	**var** e, head: List head← NULL		Two lists are created (Node* pointer)

e = malloc(
sizeof(Node));
e**->data=1**;
e**->next= NULL**;
	Allocation(e,1)
	e^.data←1
	e^.next←NULL
			A new "e" element has been created and its fields have been initialized.

head = e;	head ←e		Here, "e and head" have the same address, which means they refer to the same element.

e = malloc(
sizeof(Node));
e->data=2;
	Allocation(e,1)
	e^.data←2
			A new "e" element has been created and can be added to the top of the list by linking it to the first item in the list with its next field.
			e->next=head; head=e;
			or at the end
			head->next=e

Note in C++

```
e = malloc( sizeof(Node));⇔ e = new Node;
```

6. Operations on Linked Lists

Now we'll create a set of subroutines to manage lists, such as adding or removing an item, dend iflaying all items in the list, searching the list, and so on. It is recommended that you combine all of these functions in a dedicated list library.

Observation

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

- By using functions that take a list as a parameter and return a list. In this case, the list can be passed by value.
- By using procedures and an auxiliary element (sentry) to avoid passing by address. In this case, the list can be passed by value.
- Using unaided procedures. In this case, the list must be passed by address.

- By using functions that take a list as a parameter and return a boolean value (bool) to tell us 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. Dend iflaying a list

```
void dend iflay_list(List head) {
  while (head != NULL) {
    printf("%d->", head->data);
    head = head->next;
  }
  printf("fin\n");
}
```

```
procedure dend iflay_list(List
head)
Begin
  while (head ≠ NULL) do
    write(head->data,"->")
    head ← head^.next
  end while
  printf("end")
end
```

```
void dend iflay_list(List head) {
  if (head){
      printf("%d->", head->data);
      dend iflay_list(head->next);
  } else printf("fin\n");
}
```

We iterate through each item in the list and dend iflay the associated data. We note here that the list has been passed by value, and so the head of the original list won't be changed if we change the value of head, so we use it to browse the list safely.

6.2. List size

Go through the list and add 1 until we get to NULL

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

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

```
function size_list(List
head): integer
  var n:integer
Begin
  n←0
  while (head ≠ NULL) do
    n←n+1
    head ← head^.next
  end while
  size_list←n
end
```

6.3. Add an item to the list

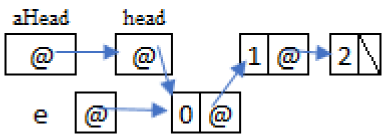
The process of adding an item to a linked list takes place in 3 steps:

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

Add an item to the top of the list (at the top)

1. Adding an item to the beginning of the list requires changing the head of the list, so it's important to pass the list by address so that this change is visible in the caller.
2. Create a new item, and if it fails, return false to the caller
3. Initialize Item

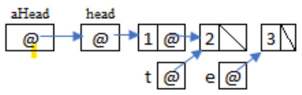
4. Replace "next" with "e" to point to the first item in the list
5. Change the head of the list to point to the new item, "aHead and e" are two local variables that are removed immediately after the procedure is executed



`int add_head(List* aHead, int d){`	`function add_head(aHead:^List,`	1
	`d:integer):bool`	
	`  var e:List`	
	`Begin`	
`  List e = malloc(sizeof(Node));`	`  allocate(e,1)`	2
`   if (e == NULL) {`	`  if (e = NULL) then`	
`     return 0;`	`    add_head ←false`	
`   }`	`  else`	
`   e-> data = d;`	`    e^.data←d`	3
`   e-> next = *aHead;`	`    e^.next←aHead^`	4
`  *aHead=e;`	`    aHead^←e`	5
`   return 1;`	`    add_head ←true`	
`}`	`  end if`	
	`end`	

Add an item at the end

1. It's possible that we'll add an element in the header, so we need to pass it by address. Create a new item
2. Initialize the element and set NULL to the "next" as this will be the last element
3. If the list is empty, we'll insert it in the head
4. If the list contains at least one item, look for the last item
5. Insert Item Last



`int append_end(List*aHead, int d){`	`function append_end(aHead:^List,`	1
`   List t;`	`d:integer): bool`	
`  List e = malloc(sizeof(Node));`	`  var e, t : List`	
`   if (e == NULL) {`	`Begin`	
`     return 0;`	`  allocate(e,1)`	
`   }`	`  if (e = NULL) then`	
	`    append_end←false`	
	`  else`	
`   e->data = d;`	`  e^.data←d`	2
`   e->next = NULL;`	`  e^.next← NULL`	
`   if (*aHead == NULL)`	`  if (aHead^=NULL) then`	3
`     *aHead = e;`	`    aHead^←e`	
`   else {`	`  else`	4
`     t= *aHead;`	`    t←aHead^`	
`     while (t-> next != NULL)`	`    while (t^.next≠NULL) do`	
`       t= t-> next;`	`       t←t^.next`	
	`    end while`	

`    t-> next=e;`
`  }`
`  return 1;`
`}`
	`    t^.next←e`
	`  end if`
	`  append_end←True`
	`  end if`
	`end`
		5

6.4. Remove an item from the list

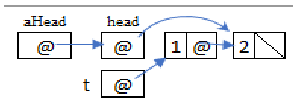
The process of removing a node from a list takes place in 4 steps:

1. Determine the previous node of the node you want to delete.
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 you want to delete.
4. Flush the memory reserved by the node you want to delete.

So there are 3 cases, either the list is empty, contains a single item, or contains more than one item.

Delete the element from the beginning (the head)

1. It's possible that we'll remove an item from the header, so we need to pass the list by address
2. If the list is empty, there is no item to delete, so we return false.
3. Stores the address of the first item to be deleted in t
4. Connecting the head with the second element
5. Remove the memory reserved by the first item



`int delete_head(List*aHead) {`
`  List t;`
	`function delete_head(aHead:^List): bool`
	`  var t:List`
	`Begin`
		1

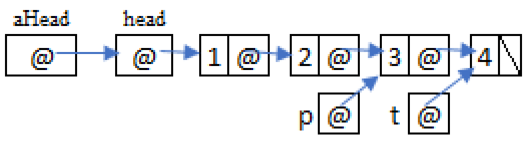
`  if (*aHead== NULL)`
`    return 0;`
	`  if (aHead^ ==NULL) then`
	`    delete_head← false`
	`  else`
		2

`  t = *aHead;`	`  t←aHead^`	3
`  *aHead =t-> next;`	`  aHead← t^.next`	4

`  free(t);`
`  return 1;`
`}`
	`  deallocate (t)`
	`  delete_head←True`
	`  end if`
	`end`	

Remove an item from the end

1. It's possible that we're removing an item from the head, so we need to pass the list by address. t is the last element and p is the second-to-last element
2. If the list is empty, there is no item to delete, so we return false
3. If there is only one item in the list, remove it directly from the head
4. If the list contains more than one element, we look for the last element t and the second-to-



last p

5. We set NULL to "next" of the penultimate p, because it has become the last, and we remove the last t from memory.

int delete_end (List*aHead) {
  List t, p;
	function delete_end(aHead:^List): bool
	  var t, p:List
	Begin
		1

if (*aHead== NULL)
    return 0;
	if (aHead^ ==NULL) **then**
	    delete_end ←false
	  else
		2

if ((*aHead)->next ==NULL) {
    free(*aHead);
    *aHead = NULL; }
	if (aHead^.next =NULL) **then**
	    dealdeal(*aHead)
	    *aHead←NULL
		3

else {
    t = *aHead;
    while (t->next != NULL) {
      p=t;
      t= t->next;
    }
	else
	    t←aHead^
	    while (t^.next ≠ NULL) **do**
	      p←t
	      t←t^.next
	    end while
		4

p->next=NULL;
    free(t);
  }
  return 1;
}
	p^.next←NULL
	    deallocate (t)
	  end if
	  delete_end ←true
	  end if
	end
		5

6.5. Delete list

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

void delete_list(List*aHead) {
  List t;
  while(*aHead!= NULL) {
    t = *aHead;
    *aHead =t-> next;
    free(t);
  }
}
	procedure delete_ list(aHead:^List)
	  var t:List
	Begin
	  while (aHead^ ≠NULL) **do**
	    t←aHead^
	    aHead^←t^.next
	    deallocate (t)
	    end while
	end
		1

void delete_list(List*aHead) {
  while (delete_head(aHead));
}
	procedure delete_ list(aHead:^List)
	Begin
	  while (aHead^ ≠NULL) **do**
	    delete_head(aHead)
	  end while
	end
		2

6.6. Main Program (Use)

int main() {
  List head =NULL;
	Begin
	  add_head(@head, 3)

```c
  add_head(&head, 3);
  add_head(&head, 2);
  append_end(&head, 4);
  add_head(&head, 1);
  append_end(&head, 5);
  printf("size=%d\n", size_list(head));
  dend iflay_list(head);
  delete_head(&head);
  delete_end(&head);
  printf("size=%d\n", size_list(head));
  dend iflay_list(head);
  delete_list(&head);
  printf("size=%d\n", size_list(head));
  dend iflay_list(head);
  return 0;
}
```

```
  add_head(@head, 2)
  append_end(@head, 4)
  add_head(@head, 1)
  append_end(@head, 5)
  write("size=", size_list(head))
  dend iflay_list(head)
  delete_head(@head)
  delete_end(@head)
  write("size=", size_list(head))
  dend iflay_list(head)
  delete_list(@head)
  write("size=", size_list(head))
  dend iflay_list(head)
end
```

- The program will dend iflay

  size=5

  1->2->3->4->5->end
- Then it will dend iflay

  size=3

  2->3->4->end
- At the end it will dend iflay

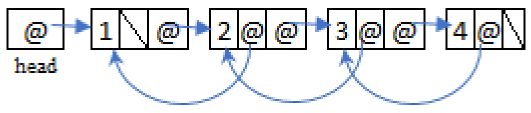
  size=0

  end

7. Double linked list

In addition to the data and the pointer that points to the next item, a doubly linked list contains another pointer, usually called a "prev," that points to the previous item. This pointer makes it easier to navigate through the list in both directions, simplifying the process of removing or inserting an item before the selected one.

The following figure shows the structure of a doubly linked list



7.1. Declaration

```c
typedef struct Node {
  int data;
  struct Node* next, * prev;
  } Node;
```

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

"data" represents the data stored in the list. "Next" is a pointer that contains the address of the next item, while "Prev" is a pointer that contains the address of the previous item.

The add and remove operations are as follows

7.2. Add an element at the beginning (header)

`e-> next = *aHead;`	`e^.next←aHead^`	1

`e-> prev = NULL;`
`if``(*aHead!= NULL)`
`  (*aHead)->prev = e;`
	`e^.prev←NULL`
	`if``(*aHead ≠ NULL)`**`then`**
	`  aHead^.prev←e`
	`END IF`
		2

`*aHead=e;`	`aHead^←e`	3

1. Change "next" from "e" to point to the first element
2. It points to NULL (first element)The "prev" of the first element, if it exists, points to the new element.
3. Change the head of the list to point to the new item

7.3. Add an item at the end

`e-> next = NULL;`	`e^.next← NULL`	1

`if`` (*aHead == NULL) {`
`  e-> prev = NULL;`
`  *aHead = e;`
`}`
	`if`` (aHead^=NULL) `**`then`**
	`  e^.prev←NULL;`
	`  aHead^←e`
		2

`else`` {`
`  t= *aHead;`
`  while (t-> next != NULL)`
`    t= t-> next;`
	`else`
	`  t←aHead^`
	`  `**`WHILE`**` (t^.next≠NULL) `**`do`**
	`    t←t^.next`
	`  `**`END WHILE`**
		3

`  E-> prev = t;`
`  t-> next=e;`
`}`
	`  e^.prev←t;`
	`  t^.next←e`
	`END IF`
		4

1. NULL because it will be the last element
2. In case the list is empty, it is appended in the header, while prev says NULL
3. If the list contains at least one item, the last item is searched for
4. The prev of the new item refers to the last item in the list. Insert Item Last

7.4. Delete the element from the beginning (the head)

1. Stores the address of the first item to be deleted
2. Bind with the second element. If the list is not empty, the "prev" of the first item must be NULL
3. Flush the memory reserved by the first item

`t = *aHead;`	`t←aHead^`	1

`*aHead =t-> next;`
`if``(*aHead!= NULL)`
`  (*aHead)->prev = NULL;`
	`aHead← t^.next`
	`if``(*aHead ≠ NULL)`**`then`**
	`  aHead^.prev← NULL`
	`END IF`
		2

`free(t);`	`deallocate (t)`	3

7.5. Remove an item from the last

1. In case the list contains more than one item, the last item t is searched, and there is no need to save the second-to-last one because it is accessible.
2. We get the second-to-last one by means of "prev" of the last t. We set "next" to NULL of the second-to-last because it has become the last. We remove the last t from memory

`t = *aHead->next;`	`t←aHead^`	1

`while (t->next!= NULL)`
`  t= t->next;`
	`WHILE  (t^.next ≠ NULL) do`
	`  t←t^.next`
	`END WHILE`	

`p=t->prev;`
`p ->next=NULL;`
`free(t);`
	`p←t^.prev`
	`p^.next←NULL`
	`deallocate (t)`
		2

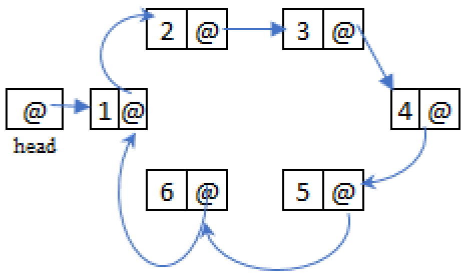
7.6. Remark:

The "prev" of the first item can be used to refer to the last item in the list, speeding up the process of accessing the last item for addition or deletion.

8. Special Linked Lists

In addition to linked single and double lists, there are linked single and double circular lists

The circular list is a normal linked list, except that the last item is not NULL, but refers to the first item in the list, as is the case in the double circular list, where the "next" of the last item refers to the first item and the "prev" of the first item refers to the last item.

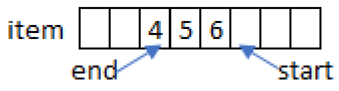


8.1. Queues

The **queue** is an abstract data structure that is used to store a set of records of the same type. It offers two essential operations: the addition of a new element, also known as an enQueue, in French: enfiler, and the deletion of an element, known as a deletion (in English: deQueue, in French: défiler). This structure respects the FIFO (First In First Out) property, which means that the first item added is the first item to be deleted. In other words, the output order is the same as the input order.

Example: list of events, queue, list of files sent to the printer, etc.

A queue can be implemented using an array using two indices to track the position of the head (or "start") and the "end" (or "end" in English). When an item is added to the queue, it is placed at the queue position and the queue index is incremented. Similarly, when an item is removed from the queue, it is removed from the head position and the end hint is also incremented. If the end of the array is reached, you can go back to the beginning of the array to continue adding items if slots are available, or you can allocate a new, larger array and copy the existing items to it.



8.1.1. Using the arrays

1. Declaration: A structure is created that contains a dynamically allocated table of items in memory, a start location "start" for adding, an "end" location for deletion, and "capacity" that contains the number of items that can be added to the table.
2. init(): The table is created and set to -1 to start and end to indicate that the queue is empty. If the creation process fails, the function returns false.
3. isEmpty(): The queue is empty if "start" and "end" are -1.

4. isFull(): The queue is full if the value of "start+1" is the same as the value of "end", and we use **mod "%" if we reach the end of the table to bring it back to the beginning.**
5. enQueue(): Makes sure the queue is not full, then adds 1 to start and adds x to the table.
6. deQueue(): Returns the first element of the array to which end points, and adds 1 to end. In case the queue is empty, it informs the user.

Note:

Normally, when a function encounters an error or unexpected behavior situation, it is not expected to return a value that could be erroneous or misinterpreted by the calling function. Instead, it must throw an exception (error) that will be caught and handled in the calling function or in another function in the call stack. By explicitly flagging the error, the exception helps identify the source of the problem and makes it easier to find and resolve the problem.

In C++ you can write

```
if(isEmpty(Q)) throw -1;
```

`typedef struct Queue{`
  `int *item;`
  `int start, end, capacity;`
`}Queue;`
	`Structure Queue`
	  `item:^integer`
	  `start, end, capacity: integer`
	`end_structure`
		1

`bool init(Queue *Q, int capacity) {`
  `Q->start = -1;`
  `Q->end = -1;`
  `Q->capacity= capacity;`
  `Q->item=`
`(int*)malloc(sizeof(int)*capacity);`
  `return Q-> item != NULL;`
`}`
	`function`
	`init(Q:^Queue,capacity:integer):bool`
	`Begin`
	  `Q^.start ←-1`
	  `Q^.end ←-1`
	  `Q^.capacity = capacity;`
	  `allocate(Q^.item, capacity)`
	  `init← Q^.item ≠ NULL`
	`end`
		2

`bool isEmpty(Queue Q){`
   `return  Q.start ==-1 && Q.end==-1;`
`}`
	`function isEmpty(Q: Queue):bool`
	`Begin`
	 `isEmpty← Q.start =-1 and Q.end=-1`
	`end`
		3

`bool isFull(Queue Q){`
 `return`
    `(Q.start+1)% Q.capacity == Q.end;`
`}`
	`function isFull(Q:Queue):bool`
	`Begin`
	 `isFull←`
	 `(Q.start+1) mod Q.  capacity = Q.end`
	`end`
		4

`void enQueue (Queue Q,int x){`
  `if(isFull(Q)){`
    `printf("error: Queue is full");`
    `return;`
  `}`
  `if(isEmpty(Q))`
   `Q.start=Q.end=0;`
  `else`
   `Q.start= (Q.start +1)% Q.capacity;`
  `Q.item[Q.start]=x;`
`}`
	`procedure enQueue(Q:Queue,`
	`x:integer)`
	`Begin`
	 `if(isFull(Q))then`
	  `write("error: Queue is full")`
	 `else`
	  `if isEmpty(Q) then`
	    `Q.start←0 Q.end←0`
	  `else`
	    `Q.start←(Q.start+1) mod`
	`Q.capacity`
	  `END IF`
	  `Q.item[Q.start] ←x`
	 `END IF`
	`end`
		5

int deQueue(Queue Q){
  if(isEmpty(Q)){
    printf("error: Queue is empty");
    return;
  }
  int x= Q.item[Q.end];
  if (Q.start == Q.end)
    Q.start = Q.end= -1;
  else
Q.end = (Q.end+1)% Q.capacity;
  return x;
}
	Function deQueue (Q: Queue): Integer
	Begin
	  if(isEmpty(Q))then
	    write("error: Queue is empty")
	    deQueue ←-1
	  else
	    deQueue←Q.item[Q.end]
	    if Q.start = Q.end **then**
	     Q.start←-1 Q.end←-1
	    else
	     Q.end←(Q.end+1) **mod** Q.capacity
	    END IF
	  END IF
	end
		6

8.1.2. Using Linked Lists:

A simple queue implementation using arrays can cause performance issues if the queue is large or heavily used, as each insertion or deletion may require moving any remaining items in the array to maintain FIFO ownership. To avoid these problems, it's best to use higher-performance data structures such as linked lists.

To simulate a queue using lists, it is necessary to add and remove items at two different ends of the list. For example, you can add new items at the end of the list and remove items at the beginning of the list. This approach can also be reversed, by adding items at the beginning of the list and removing items at the end. In both cases, the list structure allows for quick and efficient insertions and deletions, without the need for costly item moves as with the array implementation.

1. Declaration: We create a structure that contains two fields, the first referring to the first item in the list and the second referring to the last item in the list.
2. init(): by assigning a null to first and last.
3. isEmpty(): An empty Queue is an empty list.
4. enQueue():is the same as the "append_end" function, and to avoid going through all the items in the list to get to the last item, we always store the address of the last item in Q.last. In the case where the list is empty, we add the new element to the start and last, but if it is not empty, we paste the new element with the last element, and then change last so that it points to the new element.
5. deQueue():is the same as the "delete_head" function, except that the deQueue function returns the element that was deleted. So before we delete element t, let's save t-> data in x, and then delete it and return the value of x.

typedef struct Queue{
  struct Node***** first, *****last;
  int size;
} Queue;
	Structure Queue
	  first, last:^Node
	  Size :Integer
	end_structure
		1

Queue * init (){
Queue *Q=**new** Queue;
 Q->first =NULL;
 Q->last =NULL;
 Q->size =0;
 return Q
}
	procedure init (Q: ^Queue)
	Begin
	   Q^.first← NULL
	   Q^.last← NULL
	end
		2

```
bool isEmpty(Queue *Q){
   return Q->size==0;
}
```
	```
	function isEmpty(Q: Queue):bool
	Begin
	   isEmpty← Q.first= NULL
	end
	```
		3

```
void enQueue(Queue *Q, int x) {
 Node *e = malloc(sizeof(Node));
  e->data = x;
  e->next = NULL;
  if(isEmpty(Q))
    Q->first =e;
  else
    Q->last->next=e;
  Q->last =e;
  Q->size++;
}
```
	```
	procedure enQueue(var Q: Queue,
	x:integer)
	   var e:^Node
	Begin
	  allocate(e,1)
	  e^.data← x
	  e^.next← NULL
	  if (isEmpty(Q)) then
	    Q.first←e
	  else
	    Q.last^.next←e
	  END IF
	  Q.last←e
	end
	```
		4

```
int deQueue(Queue *Q) {
  Node* t; int x;
  if(isEmpty(Q)){
    printf("error: Queue is empty");
    exit(1);
  }
  t = Q->first;
  x = t ->data;
  Q->first = t-> next;
  free(t);
  return x;
}
```
	```
	function ofQueue(var Q: Queue):
	integer
	  var t:^Node
	Begin
	  if (isEmpty(Q)) then
	    printf("error: Queue is empty");
	    deQueue←-1
	  else
	    t← Q.first
	    deQueue ← t^.data
	    Q.first← t^.next
	    deallocate (t)
	  END IF
	end
	```
		5

8.2. Stacks:

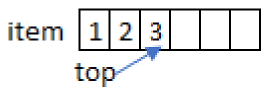
It is an abstract data structure consisting of a set of records of the same type, in which only two operations can be performed: the addition of a new element, this process is called push, and the removal of an element from the group, this process is known as pop, and these operations take place at a single end of the group called the top. This data structure has the characteristic of LIFO (Last In First Out), i.e., the last item added is the first to be removed, and the output order is therefore the opposite of the input order.

Example:

Stacks are commonly used in situations where data needs to be processed in reverse order to the order in which it was received, such as:

- Memory Management in Computer Systems
- The web browser saves the list of visited pages in a stack.
- The list of operations in Word, for example, is stored in a stack and is used to undo changes.

Stacks can be implemented using arrays or linked lists, but linked lists are often preferred because they provide more predictable performance when adding or removing items.



8.2.1. Using the table

1. Declaration: A structure is created that contains a dynamically allocated item element table, and top the add or remove location and capacity that represents the size.
2. init(): The table is created and set to top to 0 to indicate that the stack is empty.
3. isEmpty():The stack is empty if the top value is 0.
4. isFull(): If the array is full, top equals capacity.
5. Pop(): The Pop function allows you to decrement 'top' and return the last element it points to.
6. Push (): The Push function allows you to add the x element to the table and increment the 'top' pointer by 1. You need to make sure that the stack (table) is not full.

typedef struct Stack{
  int *item;
  int top, capacity;
} Stack;
	structure Stack
	  item:^integer
	  Top, size:integer
	end_structure
		1

bool init(Stack *s, int capacity) {
  s->top = 0;
  s-> capacity = capacity;
  s->item=malloc(sizeof(int)*size);
  return s->item != NULL;
}
	function init(s:^ Stack, capacity:integer):bool
	Begin
	  s^.top←0
	  s^. capacity ← capacity
	  allocate(s^.item, capacity)
	  init← s^.item ≠ NULL
	end
		2

bool isEmpty(Stack s){
   return s.top==0;
}
	function isEmpty**(s:Stack):bool**
	Begin
	   init←s.top=0
	end
		3

bool isFull(Stack s){
   return s.top==s.capacity;
}
	isFull(s:Stack):bool function
	Begin
	   isFull←s.top= s.capacity
	end
		4

int Pop(Stack s){
  int x;
  if(isEmpty(s)){
    printf("error: Stack is empty");
    exit(1);
  }
  s.top--;
  x=s.item[s.top];
  return x;
}
	function Pop(s: Stack): integer
	Begin
	  if(isEmpty(s))**then**
	    write("error: Stack is empty");
	    Pop←-1
	  else
	    s.top← s.top -1
	    Pop←s.item[s.top]
	  END IF
	end
		5

void Push(Stack s,int x){
  if(isFull(s)){
    printf("error: Stack is full");
    exit(1);
  }
  s.item[s.top]=x;
  s.top++;
}
	procedure Push(s: Stack, x: integer)
	Begin
	  if (isFull(s)) **then**
	    write("error: Stack is full");
	  else
	    s.item[s.top] ←x
	    s.top← s.top +1
	  END IF
	end
		6

8.2.2. Using Linked Lists:

To simulate a stack using lists, the addition and deletion must be done on the same side (at the beginning or at the end).

1. isEmpty():An empty stack is an empty list.
2. Pop(): The pop function is the same as the delete_head function except that the pop function returns the item that was deleted. So before we delete the first element t, we save t-> data in x, and then delete it and return the value of x.
3. Push():The push function is the same as the add_head function

bool isEmpty(List head){
   return head==NULL;
}
	function isEmpty(head: List):bool
	Begin
	   isEmpty ← head= NULL
	end
		1

int Pop(List*aHead) {
  List t; **int** x;
  if(*aHead==NULL){
    printf("error: Stack is empty");
    exit(1);
  }
  t = *aHead;
  *aHead =t-> next;
  x=t->data;
  free(t);
  return x;
}
	function Pop(aHead:^List): integer
	  var t:List
	Begin
	  if(isEmpty(aHead^))then
	    write("error: Stack is empty");
	    Pop←-1
	  else
	    t←aHead^
	    aHead← t^.next
	    Pop← t^.data
	    deallocate (t)
	  END IF
	end
		2

void Push(List* aHead, int x) {
  List e = malloc(sizeof(Node));
  e-> data = x;
  e-> next = *aHead;
  *aHead=e;
}
	procedure Push(aHead:^List, x:integer)
	   var e:List
	Begin
	  allocate(e,1)
	  e^.data← x
	  e^.next←aHead^
	  aHead^←e
	end
		3