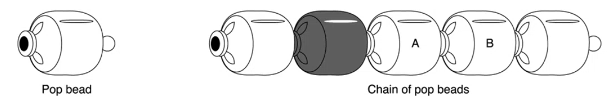
***LINKED LISTS (SINGLE LINK LISTS - 1 ADDRESS FOR EACH ENTRY)***

* A sequence of nodes
  + Each node is connected to the following one
  + Like pop beads
    - A chain can be formed easily
    - Modified easily (remove and insert)



Advantages:

* If you hold one of the beads, you can go to any entry.
* When you remove one of the beads, you are freeing that entry. You remove 2 connections and connect 2 remaining beads. That is easier than removing an entry from an array.
* (Sorted) Insertion is also simple. You just unconnect the beads that you insert your entry between. You just make 2 connections.

Disadvantages:

* You can reach third entry in array with “a[2]”. In linked lists it is not that simple. You need to start from the beginning (it’s the only thing that you know right now), then you need to go to the next one, then next one. You need to ask where the beads connected to one by one.

\*\*You can only touch heap part by yourself and stack is controlled by the program (run-time engine).

Node Structure

* Structure with pointer components
* Allocate a node as necessary
  + And connect then to form a linked list

typedef struct node\_s {

char current[3];

int volts;

struct node\_s \* linkp; 🡪 points at this struct itself.

}node\_t;

* + If we didn’t have struct node\_s definition, we can’t define linkp. We can’t do “node\_t \* linkp” because compiler wants to know what node\_t is. node\_t isn’t defined yet in the structure.
  + A structure cannot contain a member of its own type because if this is allowed then it becomes impossible for compiler to know size of such struct. Although a pointer of same type can be a member because pointers of all types are of same size and compiler can calculate size of struct.
  + Use a structure tag.

|  |
| --- |
| {current[3]} |
| {volts} |
| {address} |

|  |
| --- |
| {current[3]} |
| {volts} |
| {address} |

|  |
| --- |
| {current[3]} |
| {volts} |
| {invalid address} |

Invalid address (like 0) is for ending the linked list. malloc or calloc will never lead you to 0. It returns 0 if it fails to allocate memory.

Multiple Pointers to the Same Structure

node\_t \*n1\_p, \*n2\_p, \*n3\_p;

n1\_p = (node\_t \*)malloc(sizeof(node\_t));

n1\_p -> volts = 115;

n2\_p = (node\_t \*)malloc(sizeof(node\_t));

n2\_p -> volts = 12;

n3\_p = n2\_p;

strcpy(n1\_p->current, “AC”);

strcpy(n2\_p->current, “DC”);

n1\_p->linkp = n2\_p;

n2\_p->linkp = (node\_t \*)malloc(sizeof(node\_t));

n2\_p->linkp->volts = 220; 🡪 same as: \*(n2\_p->linkp).volts

strcpy(n2\_p->linkp->current, “AC”);

n2\_p -> linkp -> linkp = 0;

|  |
| --- |
| AC\0 |
| 115 |
| {n2\_p} |

NODE

|  |
| --- |
| {n1\_p} |

|  |
| --- |
| {n2\_p} |

|  |
| --- |
| DC\0 |
| 12 |
|  |

|  |
| --- |
| AC\0 |
| 220 |
| 0 |

|  |
| --- |
| {n3\_p} |

I can only access to whole list with n1\_p.

n1\_p is the ROOT or HEAD of our list.

List is empty when 🡪 When the list head is null (0).

Insert an Entry to the Linked List

New entry 🡪 DC, 24 Volts

You can put this new entry:

* in the beginning (inserting at the head) ---> simplest one
* to the end (tail)
* between the nodes

Insert to the head

node\_t \*t;

t = …malloc…;

t -> linkp = n1\_p;

n1\_p = t;

Insert to the end

node\_t \*t;

t = …malloc…;

t -> linkp = NULL;

n1\_p -> linkp -> linkp -> linkp = t;

------------------------------------OR---------------------------------

/\* A function that returns the last entry of a given linked list \*/

node\_t \* find\_last\_entry(node\_t \*head){

node\_t \*r;

if (head == NULL) r = NULL;

else if (head->linkp == NULL) r = head;

else r = find\_last\_entry(head->linkp);

return r;

}

node\_t \*head, \*last;

...

last = find\_last\_entry(head);

typedef struct node{

int data;

struct node \* next;

}node;

...

node \* l = NULL; 🡪 NULL means 0, convention to indicate it is not valid address

l = (node \*)malloc(sizeof(node));

l->data = 10;

l->next = NULL;

|  |
| --- |
| 10 |
| 0x0000 |

l

|  |
| --- |
| NULL |

node \* c = l;

c->next = (node \*)malloc(sizeof(node));

c = c->next;

c->data = 14;

c->next = NULL;

|  |
| --- |
| 10 |
| 0xFF00 |

|  |
| --- |
| 14 |
| 0x0000 |

l

|  |
| --- |
| NULL |

c

c->next = (node \*)malloc(sizeof(node));

c = c->next;

c->data = 14;

c->next = NULL;

|  |
| --- |
| 10 |
| 0xFF00 |

|  |
| --- |
| 14 |
| 0XFF10 |

l

|  |
| --- |
| 14 |
| 0X0000 |

|  |
| --- |
| NULL |

c

c->next = (node \*)malloc(sizeof(node));

c = c->next;

c->data = 14;

c->next = NULL;

|  |
| --- |
| 10 |
| 0xFF00 |

|  |
| --- |
| 14 |
| 0XFF10 |

l

|  |
| --- |
| 14 |
| 0XFF20 |

|  |
| --- |
| 14 |
| 0X0000 |

c

|  |
| --- |
| NULL |

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

typedef struct node{

int data;

struct node \* next;

}node;

void ll\_print(node \* l){

while (l!=NULL){

printf(“%d\n”, l->data);

l = l->next;

}

}

----------------------------------------

/\* Recursive \*/

void ll\_print\_r(node \* l){

if (l!=NULL){

printf(“%d\n”, l->data);

ll\_print\_r(l->next);

}

}

|  |  |
| --- | --- |
| 10 | 0xFF00 |

|  |  |
| --- | --- |
| 14 | 0xFF10 |

|  |  |
| --- | --- |
| 18 | 0xFF20 |

|  |  |
| --- | --- |
| 22 | 0x0000 |

l

|  |
| --- |
| NULL |

**/\* I want to get nth entry \*/**

int get\_nth(node \* l, int n){

int i;

for (i = 0; i < n && l != NULL; i++)

l = l->next;

if (l != NULL) return l->data;

else return {error, for example -1};

}

----------------------------------------

/\* Recursive \*/

int get\_nth(node \* l, int n){

if (l == NULL) return {error, for example -1};

else if (n == 0) return l->data;

else return get\_nth(l->next, n-1);

}

**/\* I want to insert something to the end \*/**

void insert\_end(node \* l, int data){

/\* new entry \*/

node \* t = (node \*)malloc(sizeof(node));

t->data = data;

t->next = NULL;

/\* getting to the end of the list \*/

while (l->next != NULL){

l = l->next;

}

l->next = t; /\* 0x0000 becomes address of t \*/

}

/\* If l is NULL, function will fail, run-time error will be given \*/

/\* You can also do like this: \*/

void insert\_end(node \* l, int data){

if (l==NULL) {You can do something here}

while (l->next != NULL){

l = l->next;

}

l->next = (node \*)malloc(sizeof(node));

l->next->data = data;

l->next->next = NULL;

}

/\* l should be reflected to caller, caller should know that l has changed, because if l is NULL, we change the l which is pointing to the beginning.

We don’t need to do this if beginning of the list (l) is not NULL \*/

void insert\_end(node \*\*l, int k){

if (\*l == NULL){

\*l = (node \*)malloc(sizeof(node));

\*l -> data = k;

\*l -> next = NULL;

}

else{

while (\*l->next != NULL) \*l = \*l->next;

\*l->next = (node \*)malloc(sizeof(node));

\*l->next->data = data;

\*l->next->next = NULL;

}

}

insert\_end(&l, 10);

**/\* Search a data in the list \*/**

int search(node \* l, int k){

int r = 0;

while (l != NULL && !r){

if (l->data == k) r = 1;

l = l->next;

}

return r;

}

----------------------------------------

/\* Recursive \*/

int search\_r(node \* l, int k){

if (l == NULL) return 0;

else if (l->data == key) return 1;

return search\_r(l->next, key);

}

----------------------------------------

**/\* Remove the last entry \*/**

void remove\_last(node \*\* l){

if (\*l == NULL) return;

if (\*l->next == NULL){

free(\*l);

\*l = NULL; /\* If we don’t do this, caller may still think that l is a valid address but I freed it so it is not valid \*/

}

/\* finding last entry \*/

while (\*l->next->next != NULL) \*l = \*l->next;

free(\*l->next); /\* freeing last entry (22) \*/

\*l->next = NULL;

}

|  |  |
| --- | --- |
| 10 | 0xFF00 |

|  |  |
| --- | --- |
| 14 | 0xFF10 |

|  |  |
| --- | --- |
| 18 | 0xFF20 |

|  |  |
| --- | --- |
| 22 | 0x0000 |

l

|  |
| --- |
| NULL |

l

|  |  |
| --- | --- |
| 10 | 0xFF00 |

|  |  |
| --- | --- |
| 14 | 0xFF10 |

|  |  |
| --- | --- |
| 18 | 0x0000 |

l

|  |
| --- |
| NULL |

----------------------------------------

MEANING OF \*\*l

typedef struct node{

int data;

struct node \* next;

}node;

...

node \* l = NULL;

l = (node \*)malloc(sizeof(node));

l->data = 10;

l->next = NULL;

removelast(l);

|  |
| --- |
| 10 |
| 0x0000 |

l

|  |
| --- |
| NULL |

In this case I need to remove (free) the entry (10) and I need to tell the caller I freed this one, l doesn’t have anything inside. Caller needs to know that l is no longer valid.

With this one:

void removelast(node \*l)

l’s value will not change. This function is returning a pointer value.

I need to do like this:

void removelast(node \*\*l)

and call like this:

removelast(&l);

Now I can change the value l inside the function removelast.

This is what I need to do because I have only 1 entry and I need to make sure that tell the user “You cannot refer to that list, I removed (freed) it.”.

You can also do like this:

node \* removelast(node \*l)

This is not convenient because I need to worry about input and the output in different places. So first one is more clear. (You can also return an int to say that the entry is empty but user needs to check it)

Only reason that we put “\*\*” is to make sure that the caller knows about the fact that we changed the input address, it is no longer valid.

You can create a helper struct:

typedef struct{

node \* l; 🡪 linked list

int num\_entries; 🡪 how many entries I have in that linked list

}linkedlist;

Now I can move linkedlist around to represent my pointer to the list and number indicating that how many entries I have in that list.

**/\* Remove a data from the list \*/**

/\* We need to do 3 thing: find it, free it, relink \*/

node \* ll\_remove(node \* l, int k){

node \*cp, \*bp; /\* cp: entry that I am gonna be finding

bp: entry that is previous to that \*/

cp = bp = l;

/\* finding the entry that will be removed \*/

while(cp != NULL && cp->data!=k){

bp = cp;

cp = cp->next;

}

if (cp!=NULL){

if (cp == bp) l = cp->next; /\* first entry is the entry

that will be removed \*/

else bp->next = cp->next; /\* relinking \*/

free(cp); /\* freeing removed entry \*/

}

return l;

}

/\* Instead of using \*\*l as parameter, we returned the l.

We should do this because user must know that if we removed first entry, l will be changed

It is user’s responsibility to return value is kept in the pointer\*/

int a[4] = {10, 14, 18, 22};

|  |  |
| --- | --- |
| 10 | 0xFF00 |

|  |  |
| --- | --- |
| 14 | 0xFF10 |

|  |  |
| --- | --- |
| 18 | 0xFF20 |

|  |  |
| --- | --- |
| 22 | 0x0000 |

l

|  |
| --- |
| NULL |

What are the differences for int linked list and int array?

|  |  |  |
| --- | --- | --- |
|  | LINKED LIST  (n numbers) | ARRAY  (n numbers) |
| Amount of Memory | n \* (4 + 8) bytes | n \* 4 bytes |
| Random Access or  Accessing nth entry | get\_nth()  k \* (1 addition + 1 memory access) + memory lookup | a[k] 🡪 (a + k\*4)  1 multiplication + 1 addition + memory lookup |
| Growing/Shrinking by 1  Insert/Remove | requires 1 malloc | reallocating and recopying of n entries |

**/\* Inserting the entry in its right place \*/**

node \* ll\_sorted\_insert(node \* l, int k){

node \*ce, \*ne, \*n;

/\* insert at the head \*/

if (l == NULL || l->data > k){

n = (node \*)malloc(sizeof(node));

n -> data = k;

n -> next = l;

l = n;

}

else{

/\* search \*/

ce = l;

ne = ce -> next;

while(ce!=NULL && ne!=NULL && !(ce->data<k && k<ne->data)){

ce = ne;

ne = ce->next;

}

n = (node \*)malloc(sizeof(node));

ce->next = n;

n->data = k;

n->next = ne;

}

return l;

}

**/\* Sorting the list \*/**

node \* ll\_sorted\_insert(node \* l, int k){

node \*sl = NULL;

node \*t;

while (l!=NULL){

sl = ll\_sorted\_insert(sl, l->data);

t = l;

l = l->next;

free(t);

}

}

|  |  |
| --- | --- |
| 10 | 0xFF00 |

|  |  |
| --- | --- |
| 14 | 0xFF10 |

|  |  |
| --- | --- |
| 18 | 0xFF20 |

|  |  |
| --- | --- |
| 22 | 0x0000 |

l

|  |
| --- |
| NULL |

We want to make this ascending ordered list a descending ordered list

I need to go from tail to head

If I have an entry with previous field as well, I can represent this list as a double linked list.

|  |  |  |
| --- | --- | --- |
|  | 18 |  |

|  |  |  |
| --- | --- | --- |
|  | 22 | NULL |

|  |  |  |
| --- | --- | --- |
| NULL | 10 |  |

|  |  |  |
| --- | --- | --- |
|  | 14 |  |

If my list is like this:

|  |  |  |
| --- | --- | --- |
|  | 18 |  |

|  |  |  |
| --- | --- | --- |
|  | 22 |  |

|  |  |  |
| --- | --- | --- |
|  | 10 |  |

|  |  |  |
| --- | --- | --- |
|  | 14 |  |

This is double link circuler list.

typedef struct dnode{

|  |  |  |
| --- | --- | --- |
| previous | data | next |

struct dnode \* next;

int data;

struct dnode \* prev;

}dnode;

void test1(){

dnode \* l = NULL;

l = (dnode \*)malloc(sizeof(dnode));

l -> data = 10;

l -> next = NULL;

l -> prev = NULL;

}

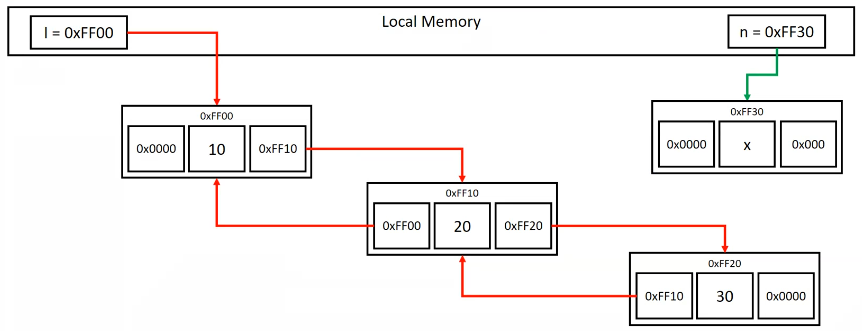
|  |  |
| --- | --- |
| LOCAL MEMORY  (test1)   |  | | --- | | l = 0xFF00 | |

|  |  |  |  |
| --- | --- | --- | --- |
| HEAP  0xFF00   |  |  |  | | --- | --- | --- | | NULL | 10 | NULL | |

On exiting the function test1, the local memory is no longer accessible by any expression of statement in the program.

If you need the allocated node, make sure that the function returns it somehow.

If you do not need it node, make sure that it is freed before exiting the function.



Insert at the end

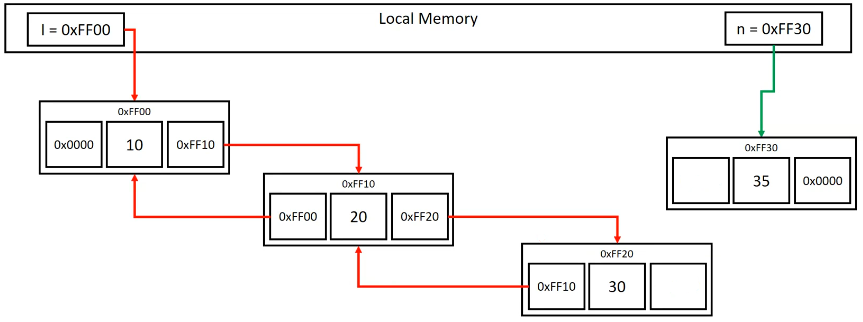
/\* Finding last entry (with data 30 in this case) \*/

t = l;

while (t -> next != NULL) t = t -> next;

/\* Inserting the entry \*/

t -> next = n;

n -> prev = t;

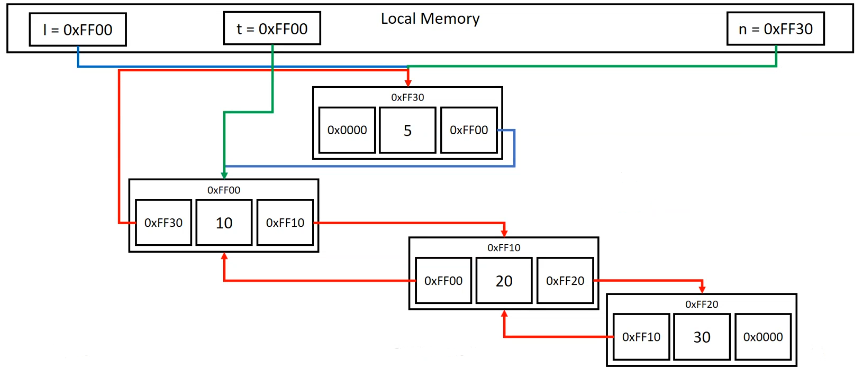
0xFF30

0xFF20

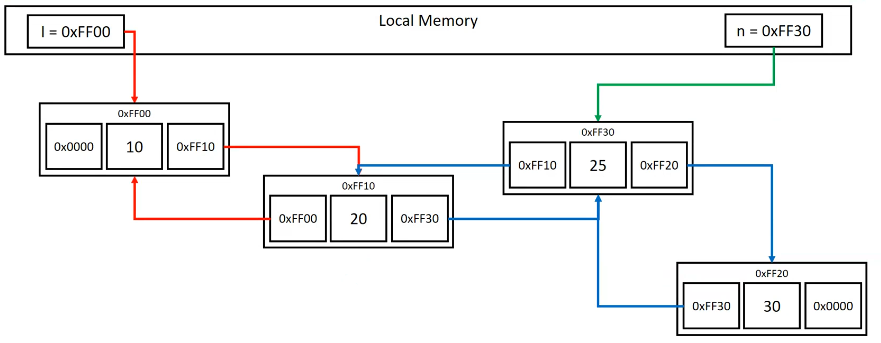
n -> next = NULL;

n -> data = 35;

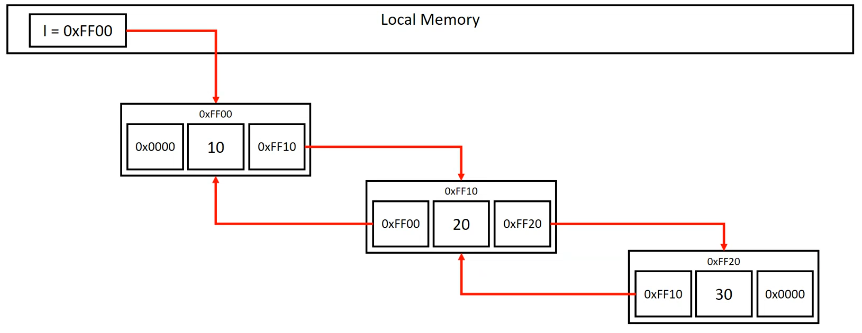
Insert at the beginning



Insert at the middle



Remove



If I remove last entry, I should only make free last entry and {0xFF10}->next = NULL

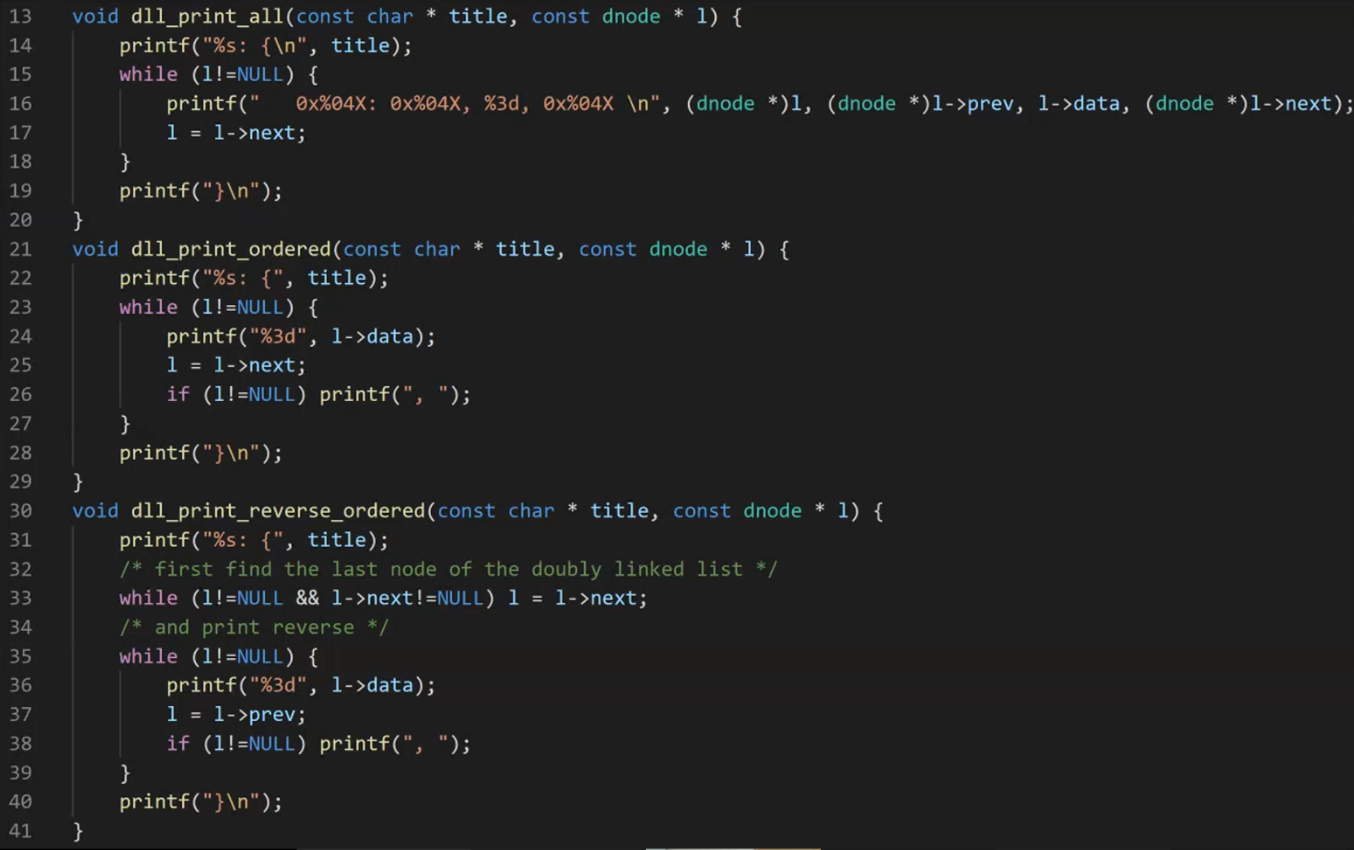
If I remove first entry, I should make free l and l = 0xFF10 and {0xFF10}->prev = NULL

If I remove 0xFF10, I should free it and {0xFF00}->next = 0xFF20, {0xFF20}->prev = 0xFF00

0xFF10->next

0xFF10->prev

When you define l, you should initialize it with NULL.



STACKS (ABSTRACT DATA TYPE) WITH LINKED LISTS

Array, linked list, stack, queue, trees, graph… are abstracted concepts that has certain properties that helps us solve some problems.

Stack is dynamic (grows and shrinks depending on the usage) list.

Access has specific ways 🡪 FILO / LIFO

You can **only** get your last entered entry.

For stack:

* insertion 🡪 push (insert at front)
* accessing/removing 🡪 peek/pop (remove from the front)

While implementing stack:

* Code should be efficient.
  + Getting the last entry should be very fast.
  + It should require a storage as much space as needed. Not necessarily more than number of entries I have.
* Ease of use.

Growing and shrinking arbitrarily for dynamic list is needed dynamic data structure.

eliminates the use of fixed size array

linked list

reallocation of an array

|  |  |  |
| --- | --- | --- |
| STORAGE | 3X | X |
| RUN-TIME EFFICIENCY | not good (O(n)) | good (just a[k]) 🡪 O(1) |

|  |  |  |
| --- | --- | --- |
| LIFO | O(1) | O(1) |
| -ALLOCATION-  GROWING/SHRINKING | good | not good |
| EASE OF USE | I just need to get the head, and I can reach wherever I want. | double \* array;  int counter; -> identify where is my last entry |

Although stack doesn’t require random access, it requires only access last element, head for linked list.

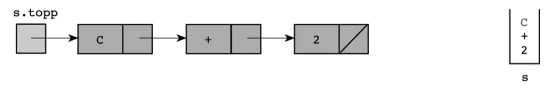
If you have limited storage, you should choose reallocation of an array,

If you favor run-time efficiency, you should choose linked list.

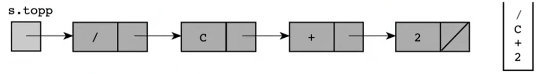
Linked List Representation of Stacks

* Stack can be implemented using array
* Stack = LIFO List
  + Linked list can be used

Stack of three characters



Stack after insertion (push) of ‘/’

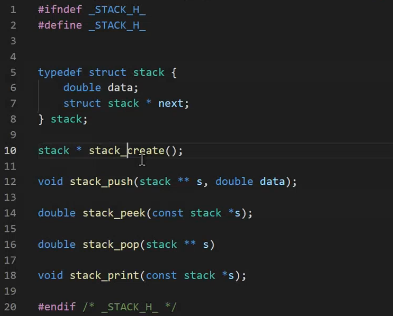


When we write a compiler (or interpreter), evaluation of expressions can be done using stacks.

STACK.H 🡪 has all the necessary information for smn to use my stack

#ifndef \_STACK\_H\_ 🡪 if \_STACK\_H\_ is defined, don’t do anything. If it is not defined, use your definitions. So if our header file (stack.h) is included more than once, ifndef prevent possible errors.

I will give user .h file and compiled version (.o) file.



;

stack\_create is for creating our stack and set it to NULL.

STACK.C

#include <stdio.h>

#include <stdlib.h>

#include “stack.h”

stack \* stack\_create(){

return NULL;

}

void stack\_push(stack \*\*s, double data){

stack \* t = (stack \*)malloc(sizeof(stack));

t->data = data;

t->next = \*s;

\*s = t;

}

double stack\_peek(const stack \*s){

double x;

if (s!=NULL) x = s->data;

return x;

}

double stack\_pop(stack \*\*s){

double x;

stack \*t = \*s;

if (t != NULL){

x = t->data;

\*s = t->next;

free(t);

}

return x;

}

void stack\_print(const stack \*s){

/\* must use just for debugging, because we are accessing randomly, it is not what stack is for \*/

printf(“Stack: ”);

while (s!=NULL){

printf(“ %f”, s->data);

s = s->next;

}

printf(“\n”);

}

TEST.C

#include <stdio.h>

#include <stdlib.h>

#include “stack.h”

int main(){

stack s = NULL;

double x;

s = stack\_create();

stack\_push(&s, 10.0);

stack\_push(&s, 20.0);

stack\_push(&s, 30.0);

stack\_print(s); 🡪 Stack: 30.000000 20.000000 10.000000

x = stack\_pop(&s);

printf(“Popped: %f\n”, x); 🡪 Popped: 30.000000

stack\_print(s); 🡪 Stack: 20.000000 10.000000

x = stack\_peek(s);

printf(“Entry at top: %f”, x); 🡪 Entry at top: 20.000000

stack\_print(s); 🡪 Stack: 20.000000 10.000000

}

void f1(int \*x){

/\* x is holding an address to an integer value in memory \*/

/\* you can use \*x to access to that value\*/

}

void f2(int \*\*x){

/\*x is holding an address of an address to an integer value in memory\*/

/\* In this case \*x means secondary address

\*(\*x) means getting the value \*/

\*x = NULL; 🡪 /\* modifies t’s value \*/

}

void main(){

int y=10;

int \*t;

t = &y; /\* t is holding y’s address \*/

f2(&t);

/\* t is no longer holding y’s address, instead it is holding NULL, it is pointing to nothing \*/

}

QUEUE (ABSTRACT DATA TYPE) WITH LINKED LISTS

* Queue = FIFO list (dynamic) not random
* EX: Model a line of customers waiting at a checkout counter
* Need to keep both ends of a queue
  + Front end rear

Again, we can use linked list or dynamically allocated arrays. We know arrays are good for storage and linked lists are good for growing/shrinking.

What about efficiency?

|  |  |  |
| --- | --- | --- |
|  | LINKED LIST | ARRAY |
| FIFO | If head is last entry (like stack):   * Insertion🡪no cost O(1) * Remove🡪problem O(n)   We can use circular double linked list. | O(1)  You can get to the end of the list very simply. |

head->next

HEAD

head->previous