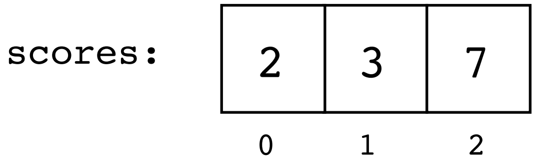
We’ve been looking at arrays:

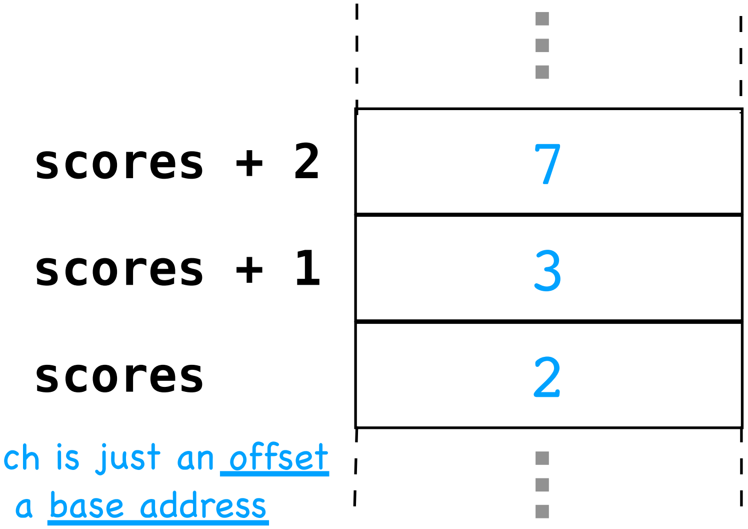
array elements are **contiguous** in memory



scores:

2	3	7
0	1	2

making it easy to access each element with an index…



scores + 2 | 7

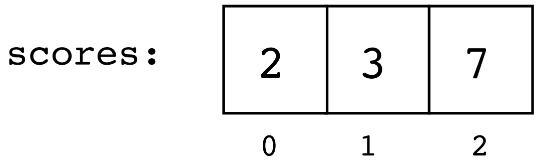
scores + 1 | 3

scores | 2

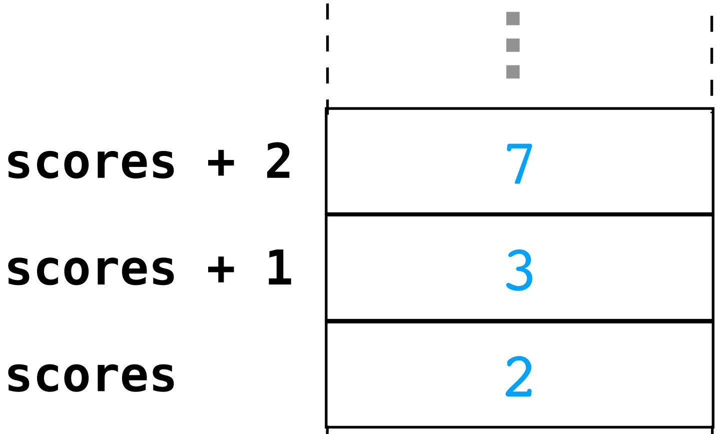
…which is just an offset from a base address

Now let’s look at linked lists…

array elements are **contiguous** in memory

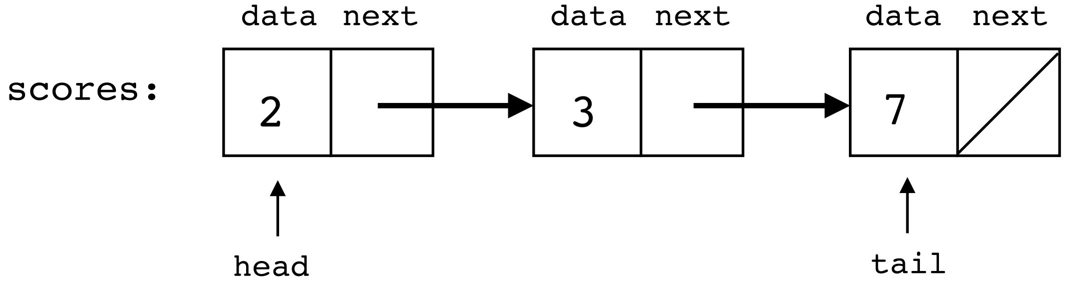


making it easy to access each element with an index...

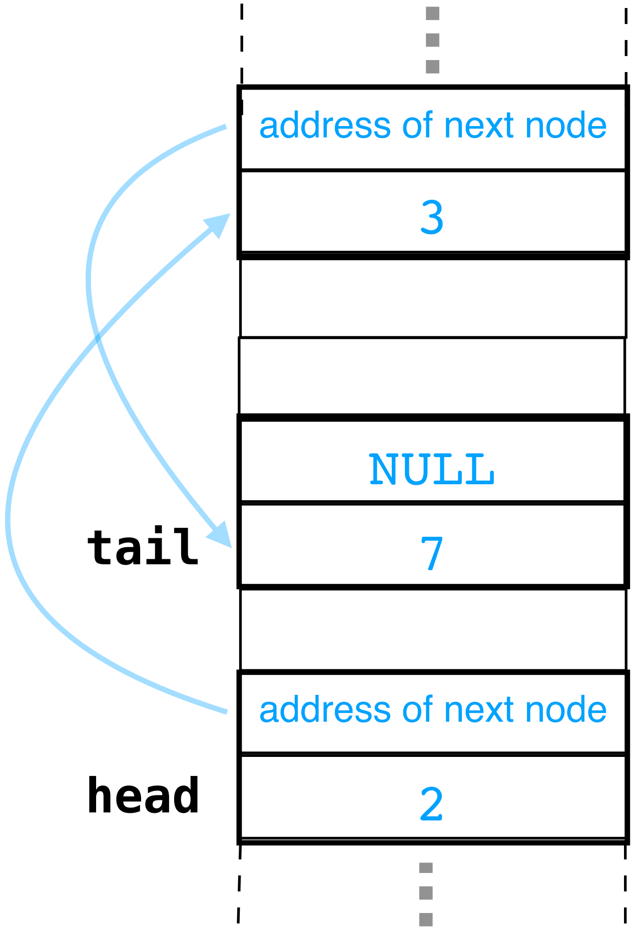


...which is just an offset from a base address

linked list elements are not likely to be contiguous in memory



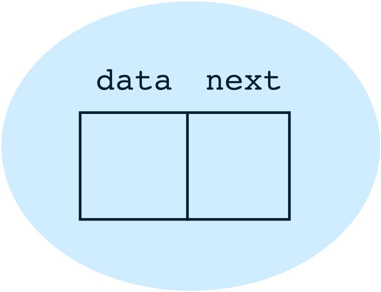
the sequence is represented by a set of nodes, each of which contains both the data and a 'link' or pointer to the next node



Linked Lists in C

```c
#include <stdio.h>
#include <stdlib.h>

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

int main()
{
    struct node* head = NULL;
    struct node* tail = NULL;

    //---- create a node and insert it at head
    head = malloc(sizeof(struct node));
    head->data = 7;
    head->next = NULL;
    tail = head;
}
```

a singly linked list



NULL
head
NULL
tail

Linked Lists in C

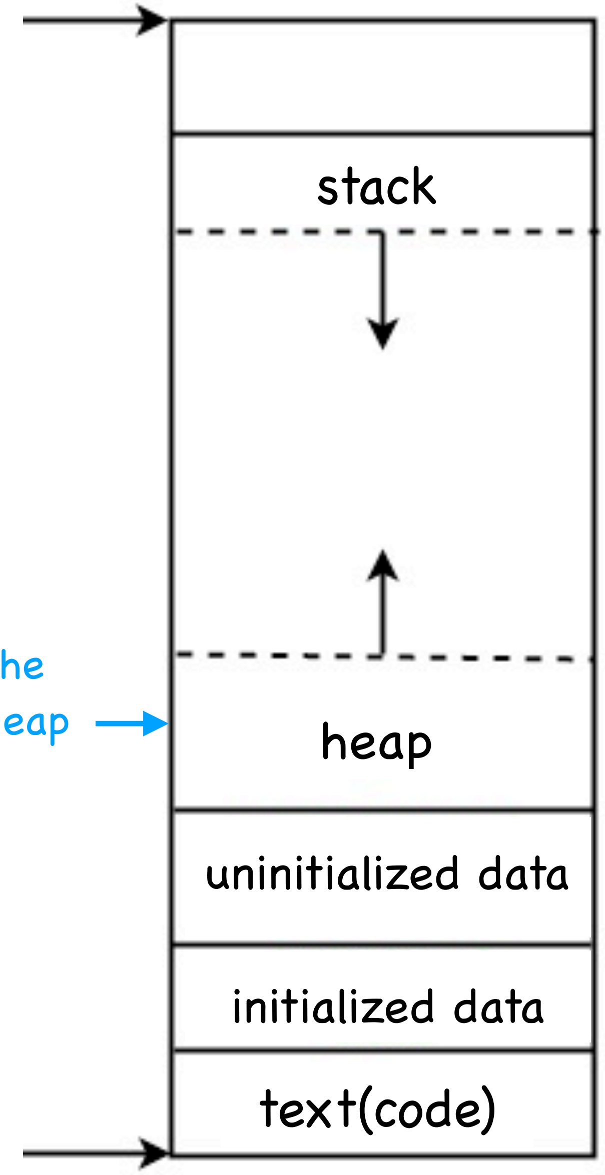
```c
#include <stdio.h>
#include <stdlib.h>

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

int main()
{
    struct node* head = NULL;
    struct node* tail = NULL;

    //---- create a node and insert it at head
    head = malloc(sizeof(struct node));
    head->data = 7;
    head->next = NULL;
    tail = head;
}
```

```c
void *malloc( size_t size );
```

Allocates `size` bytes of uninitialized memory and returns a pointer or NULL if unsuccessful.

Memory Layout (Map) for a Running C Program

highest address



local variables

`malloc()` allocates memory on the process heap. It remains on the heap until it is freed using `free()`.

dynamically allocated memory

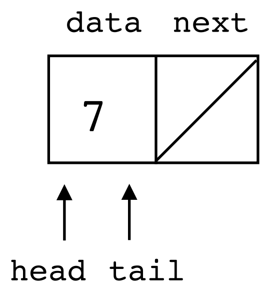
global variables

lowest address

machine instructions

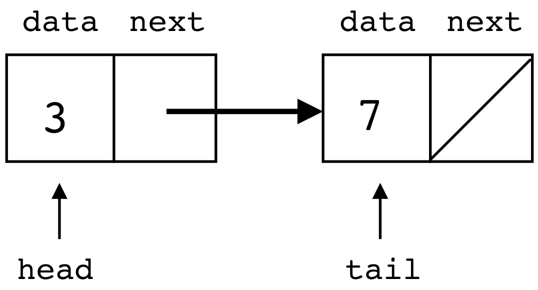
Linked Lists in C

```c
#include <stdio.h>
#include <stdlib.h>

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

int main()
{
    struct node* head = NULL;
    struct node* tail = NULL;

    //---- create a node and insert it at head
    head = malloc(sizeof(struct node));
    head->data = 7;
    head->next = NULL;
    tail = head;
}
```



Linked Lists in C

```
#include <stdio.h>
#include <stdlib.h>

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

int main()
{
    struct node* head = NULL;
    struct node* tail = NULL;

    //---- create a node and insert it at head
    head = malloc(sizeof(struct node));
    head->data = 7;
    head->next = NULL;
    tail = head;

    //---- create a node and insert it at head
    struct node* newnode;
    newnode = malloc(sizeof(struct node));
    newnode->data = 3;
    newnode->next = head;
    head = newnode;
}
```



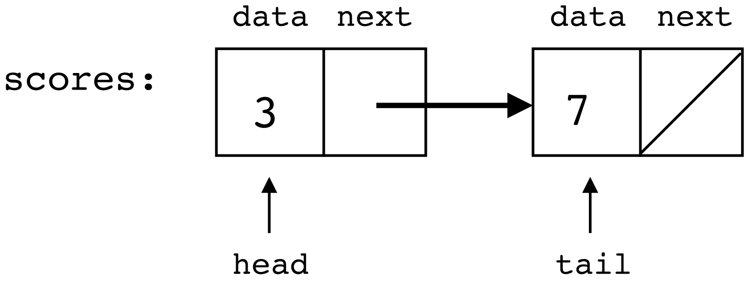
Linked Lists in C

```c
#include <stdio.h>
#include <stdlib.h>

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

int main()
{
    struct node* head = NULL;
    struct node* tail = NULL;

    //---- create a node and insert it at head
    head = malloc(sizeof(struct node));
    head->data = 7;
    head->next = NULL;
    tail = head;

    //---- create a node and insert it at head
    struct node* newnode;
    newnode = malloc(sizeof(struct node));
    newnode->data = 3;
    newnode->next = head;
    head = newnode;

    //---- traverse the list and print the data
    struct node* tmp = head;
    while(tmp != NULL)
    {
        printf("%d->",tmp->data);
        tmp = tmp->next;
    }
    printf("NULL\n");
    return 0;
}
```

```
3->7->NULL
```

Linked Lists in C

```c
#include <stdio.h>
#include <stdlib.h>

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

struct singlylinkedlist
{
    struct node* head;
    struct node* tail;
};

typedef struct singlylinkedlist slist;

int main()
{
   slist scores;

    //---- create a node and insert it at head
    scores.head = malloc(sizeof(struct node));
    scores.head->data = 7;
    scores.head->next = NULL;
    scores.tail = scores.head;

    //---- create a node and insert it at head
    struct node* newnode;
    newnode = malloc(sizeof(struct node));
    newnode->data = 3;
    newnode->next = scores.head;
    scores.head = newnode;
```

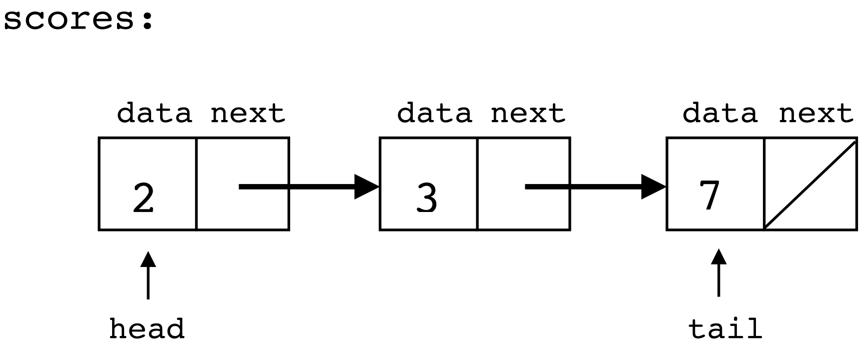


Linked Lists in C

```c
#include <stdio.h>
#include <stdlib.h>

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

struct singlylinkedlist
{
    struct node* head;
    struct node* tail;
};

typedef struct singlylinkedlist slist;

void insertHead(slist* list, int data)
{
    //---- create new node
    struct node* newnode;
    newnode = malloc(sizeof(struct node));
    newnode->data = data;
    newnode->next = list->head;

    //---- adjust pointers
    list->head = newnode;
    if (list->tail == NULL)
        list->tail = newnode;
}
```

```c
int main()
{
    slist scores = {NULL, NULL};

    insertHead( &scores, 7);
    insertHead( &scores, 3);
    insertHead( &scores, 2);

    //---- traverse and print data

    struct node* tmp = scores.head;

    while(tmp != NULL)
    {
        printf("%d->",tmp->data);
        tmp = tmp->next;
    }
    printf(“NULL\n");

    return 0;
}
```

```
2->3->7->NULL
```



```c
#include <stdio.h>
#include <stdlib.h>

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

struct singlylinkedlist
{
    struct node* head;
    struct node* tail;
};

typedef struct singlylinkedlist slist;

void insertHead(slist* list, int data)
{
    //---- create new node
    struct node* newnode;
    newnode = malloc(sizeof(struct node));
    newnode->data = data;
    newnode->next = list->head;

    //---- adjust pointers
    list->head = newnode;
    if (list->tail == NULL)
        list->tail = newnode;
}

void printList(slist* list)
{
    struct node* tmp = list->head;
    while(tmp != NULL)
    {
        printf("%d->",tmp->data);
        tmp = tmp->next;
    }
    printf("NULL\n");
}
```

```c
int main()
{
    slist scores = {NULL, NULL};

    insertHead( &scores, 7 );
    insertHead( &scores, 3);
    insertHead( &scores, 2);

    printList(&scores);
    return 0;
}
```

slist.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "slist.h"

void insertHead(slist* list, int data)
{
    //---- create new node
    struct node* newnode;
    newnode = malloc(sizeof(struct node));
    newnode->data = data;
    newnode->next = list->head;

    //---- adjust pointers
    list->head = newnode;
    if (list->tail == NULL)
        list->tail = newnode;
}

void printList(slist* list)
{
    struct node* tmp = list->head;
    while(tmp != NULL)
    {
        printf("%d->",tmp->data);
        tmp = tmp->next;
    }
    printf("NULL\n");
}
```

slist.h

```c
#ifndef slist_h
#define slist_h

#include <stdio.h>

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

struct singlylinkedlist
{
    struct node* head;
    struct node* tail;
};

typedef struct singlylinkedlist slist;


void insertHead(slist* list, int data);
void printList(slist* list);

#endif /* slist_h */
```

main.c

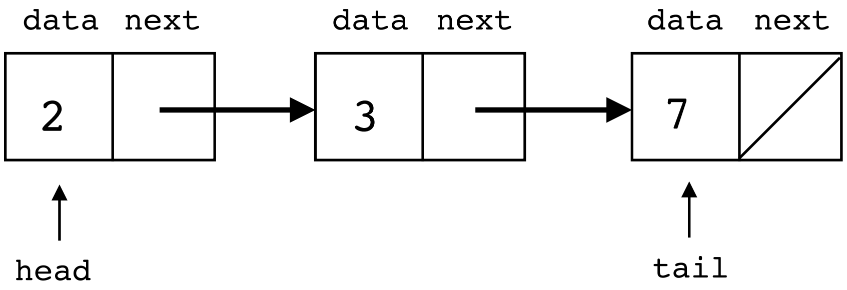
```c
#include <stdio.h>
#include <stdlib.h>
#include "slist.h"

int main()
{
    slist scores = {NULL, NULL};

    insertHead( &scores, 7 );
    insertHead( &scores, 3);
    insertHead( &scores, 2);

    printList(&scores);
    return 0;
}
```

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "slist.h"

int main()
{
    slist scores = {NULL, NULL};

    insertHead( &scores, 7 );
    insertHead( &scores, 3);
    insertHead( &scores, 2);

    printList(&scores);
    return 0;
}
```

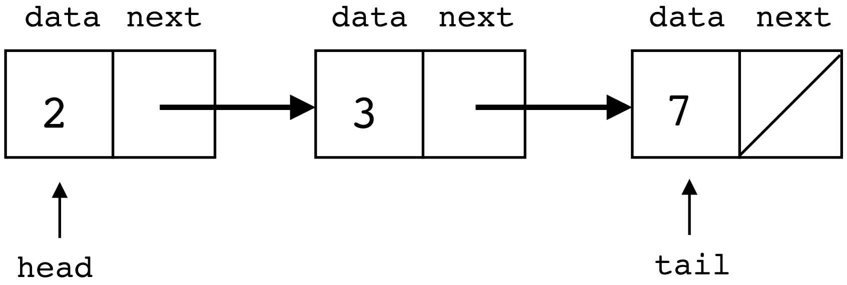
what other functions may be needed?

```
insertTail
removeHead
removeTail
remove
. . .
```

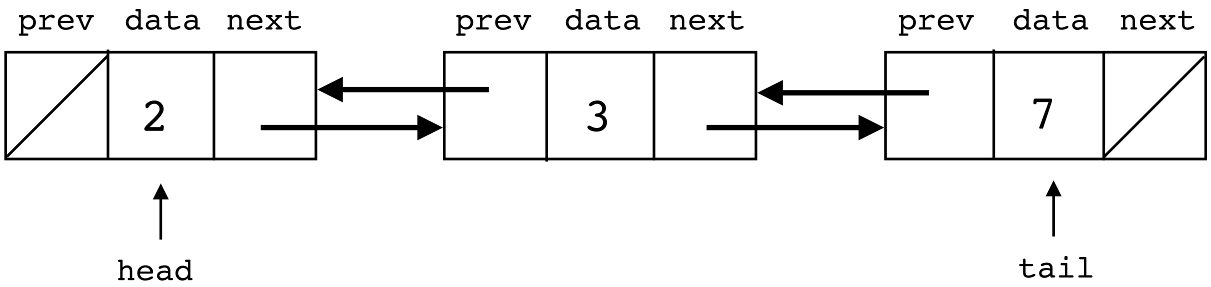


what other functions may be needed?

```
insertTail
removeHead
removeTail  ...is this easy?
remove   ...how about this?
. . .
```

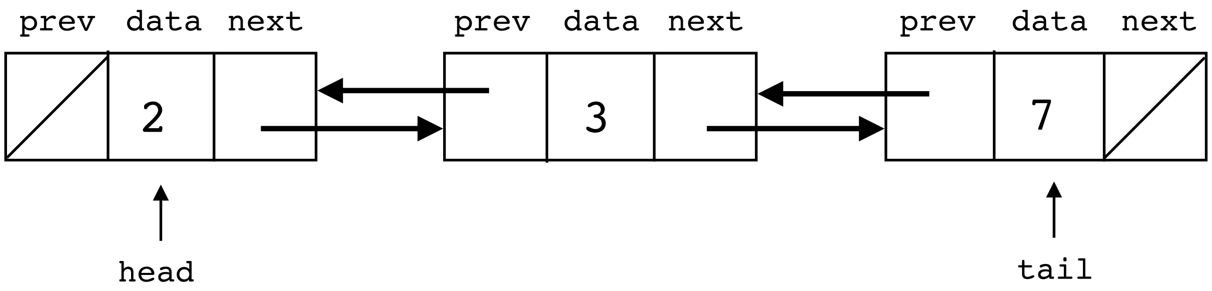


Doubly Linked Lists



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

Doubly Linked Lists



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

Advantages	Disadvantages
1. bi-directional traversal	1. increased memory usage
2. efficient deletion	2. more complex implementation
3. insertion/deletion at both ends in constant time	