Linked Lists and other Dynamic Data Structures
Arrays

Fixed in size
Allocated in advance within a contiguous memory block

Look-up is fast
Resizing and Deleting is hard (reallocate and copy)
Dynamic Data Structures

Grow and shrink one element at a time
Not allocated within contiguous memory block

Resizing and Deleting is easy
Look-up can be hard
Linked Lists
Linked Lists

A linear series of memory locations connected with pointers
Linked Lists

A linear series of memory locations connected with pointers

typedef struct _node{
    int data;
    struct node* next;
}node;
Linked Lists vs. Arrays

- Grow and shrink one element at a time
- No copying or realloc cost on extending

- Not allocated contiguously in memory
- Look-up/searching takes linear time, $O(n)$ random accesses
More on Implementation
More on Implementation

Beginning of a linked list
Keep track of the head node.

End of a linked list
    tail->next=NULL;
More on Implementation

Beginning of a linked list
Keep track of the head node.

End of a linked list
tail->next=NULL;

Doubly-linked lists
Linked lists can also be doubly-linked and have pointers to next as well as the previous nodes

typedef struct _node {
    int data;
    struct node* next;
    struct node* prev;
}node;

Data

Node

Pointer to next

Pointer to prev
Initializing a List
Initializing a List

```c
node* head=NULL;
```

head points to the beginning of the (empty) list
Initializing a List

node* head=NULL;

head points to the beginning of the (empty) list

head=malloc(sizeof(node));
head->data=1;
head->next=NULL;

head now contains the first element of the list
Initializing a List

```
node* head=NULL;
```

Head points to the beginning of the (empty) list

```
head=malloc(sizeof(node));
head->data=1;
head->next=NULL;
```

Head now contains the first element of the list

```
struct list{
    node* head;
    node* tail;  //optional
};
```

The list can be represented as a struct of head and tail pointer
Adding an Element
Adding an Element
Adding an Element

Adding to head

...  
newNode = malloc(sizeof(node));
newNode->data = 1;
newNode->next = myList->head;
myList->head = newNode;
...
Adding an Element

Adding to head

...  
newNode = malloc(sizeof(node));  
newNode->data = 1;  
newNode->next = myList->head;  
myList->head = newNode;  
...
Adding an Element

Adding to head

... 
newNode = malloc(sizeof(node));
newNode->data = 1;
newNode->next = myList->head;
myList->head = newNode;
...

H → ... → New → ...

H → New → ...

New
Adding an Element
Adding an Element
Adding an Element

Adding to tail 1

```c
node *newNode,*temp;
newNode=malloc(sizeof(node));
newNode->data=num;
temp=myList->head;

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

temp->next=newNode;
newNode->next=NULL;
```
Adding an Element

Adding to tail 1

```c
node *newNode,*temp;
newNode=malloc(sizeof(node));
newNode->data=num;
temp=myList->head;

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

temp->next=newNode;
newNode->next=NULL;
```

Adding an Element

Adding to tail 1

```c
node *newNode,*temp;
newNode=malloc(sizeof(node));
newNode->data=num;
temp=myList->head;

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

temp->next=newNode;
newNode->next=NULL;
```
Adding an Element

Adding to tail 2

```c
// Assuming we have a pointer to tail
node* newNode;
newNode = malloc(sizeof(node));

node* temp = myList->tail;
temp->next = newNode;

newNode->next = NULL;
myList->tail = newNode;
```
Adding an Element

Adding to tail 2

// Assuming we have a pointer to tail
node* newNode;
newNode = malloc(sizeof(node));

node* temp = myList->tail;
temp->next = newNode;

newNode->next = NULL;
myList->tail = newNode;
Adding an Element

Adding to tail 2

```c
// Assuming we have a pointer to tail
node* newNode;
newNode = malloc(sizeof(node));

node* temp = myList->tail;
temp->next = newNode;

newNode->next = NULL;
myList->tail = newNode;
```
// Assuming we have a pointer to tail
define newNode:
newNode=malloc(sizeof(node));

define temp=myList->tail:
temp->next=newNode;

newNode->next=NULL;
myList->tail=newNode;
Adding an Element
Adding an Element

Left → Right

Diagram showing the addition of an element from the left to the right.
Adding an Element

Adding in the middle

```c
int i;
ode* left, right, temp, newNode;
newNode = malloc(sizeof(node));
newNode->data = 1;
temp = myList->head;

for (i = 1; i < pos - 1; i++)
    temp = temp->next;
left = temp;

right = left->next;
left->next = newNode;
newNode->next = right;
```
Adding an Element

Adding in the middle

```c
int i;
node* left, right, temp, newNode;
newNode = malloc(sizeof(node));
newNode->data = 1;
temp = myList->head;
for (i = 1; i < pos - 1; i++)
    temp = temp->next;
left = temp;
right = left->next
left->next = newNode;
newNode->next = right;
```
Adding an Element

Adding in the middle

```c
int i;
node* left, right, temp, newNode;
newNode = malloc(sizeof(node));
newNode->data = 1;
temp = myList->head;

for (i = 1; i < pos - 1; i++)
    temp = temp->next;
left = temp;
right = left->next;
left->next = newNode;
newNode->next = right;
```
Traversing Linked Lists

```
node* current = myList->head;
while (current != NULL) {
    /* Do Stuff */
    current = current->next;
}
```
Deleting an Element
Deleting an Element
Deleting an Element

```c
if (to_delete == myList->head) {
    myList->head = to_delete->next;
    free(to_delete);
} else {
    left->next = to_delete->next;
    free(to_delete);
}
```
Deleting an Element

```c
if (to_delete == myList->head) {
    myList->head = to_delete->next;
    free(to_delete);
} else {
    left->next = to_delete->next;
    free(to_delete);
}
```
Deleting an Element

```c
if (to_delete == myList->head) {
    myList->head = to_delete->next;
    free(to_delete);
} else {
    left->next = to_delete->next;
    free(to_delete);
}
```
Recursive Reverse
Recursive Reverse

```c
void RecursiveReverse(node** headRef) {
    node* first;
    node* rest;

    if (*headRef == NULL) return;  // empty list base case

    first = *headRef;  // suppose first = {1, 2, 3}
    rest = first->next;  // rest = {2, 3}

    if (rest == NULL) return;  // empty rest base case
    RecursiveReverse(&rest);  // Recursively reverse the smaller {2, 3} case

    first->next->next = first;  // put the first elem on the end of the list
    first->next = NULL;

    *headRef = rest;  // fix the head pointer
}
```
```c
void RecursiveReverse(node** headRef) {
    node* first;
    node* rest;

    if (*headRef == NULL) return;

    first = *headRef;
    rest = first->next;
    if (rest == NULL) return;
    RecursiveReverse(&rest);

    first->next->next = first;
    first->next = NULL;

    *headRef = rest;
}
```
void RecursiveReverse(node** headRef) {
    node* first;
    node* rest;

    if (*headRef == NULL) return;

    first = *headRef;
    rest = first->next;
    if (rest == NULL) return;
    RecursiveReverse(&rest);

    first->next->next = first;
    first->next = NULL;

    *headRef = rest;}

rest₁ = 1
first₁ = 0
void RecursiveReverse(node** headRef) {
    node* first;
    node* rest;

    if (*headRef == NULL) return;

    first = *headRef;
    rest = first->next;
    if (rest == NULL) return;
    RecursiveReverse(&rest);

    first->next->next = first;
    first->next = NULL;

    *headRef = rest;}

rest2 = 2
first2 = 1
rest1 = 1
first1 = 0
void RecursiveReverse(node** headRef) {
    node* first;
    node* rest;

    if (*headRef == NULL) return;

    first = *headRef;
    rest = first->next;
    if (rest == NULL) return;
    RecursiveReverse(&rest);

    first->next->next = first;
    first->next = NULL;

    *headRef = rest;
}
```c
void RecursiveReverse(node** headRef) {
    node* first;
    node* rest;

    if (*headRef == NULL) return;

    first = *headRef;
    rest = first->next;

    if (rest == NULL) return;
    RecursiveReverse(&rest);

    first->next->next = first;
    first->next = NULL;

    *headRef = rest;
}
```
void RecursiveReverse(node** headRef) {
    node* first;
    node* rest;
    if (*headRef == NULL) return;
    first = *headRef;
    rest = first->next;
    if (rest == NULL) return;
    RecursiveReverse(&rest);
    first->next->next = first;
    first->next = NULL;
    *headRef = rest;}

rest2 = 2
first2 = 1
rest1 = 1
first1 = 0
```c
void RecursiveReverse(node** headRef)
{
    node* first;
    node* rest;

    if (*headRef == NULL) return;

    first = *headRef;
    rest = first->next;
    if (rest == NULL) return;
    RecursiveReverse(&rest);

    first->next->next = first;
    first->next = NULL;
    *headRef = rest;
}
```
void RecursiveReverse(node** headRef) {
    node* first;
    node* rest;

    if (*headRef == NULL) return;

    first = *headRef;
    rest = first->next;
    if (rest == NULL) return;
    RecursiveReverse(&rest);

    first->next->next = first;
    first->next = NULL;

    *headRef = rest;}

rest2 = 2
first2 = 1
rest1 = 1
first1 = 0
void RecursiveReverse(node** headRef) {
    node* first;
    node* rest;

    if (*headRef == NULL) return;

    first = *headRef;
    rest = first->next;
    if (rest == NULL) return;
    RecursiveReverse(&rest);

    first->next->next = first;
    first->next = NULL;

    *headRef = rest;
}
```c
void RecursiveReverse(node** headRef) {
    node* first;
    node* rest;

    if (*headRef == NULL) return;

    first = *headRef;
    rest = first->next;
    if (rest == NULL) return;
    RecursiveReverse(&rest);

    first->next->next = first;
    first->next = NULL;

    *headRef = rest;
}
```
void RecursiveReverse(node** headRef) {
    node* first;
    node* rest;

    if (*headRef == NULL) return;

    first = *headRef;
    rest = first->next;
    if (rest == NULL) return;
    RecursiveReverse(&rest);

    first->next->next = first;
    first->next = NULL;

    *headRef = rest;}

rest1 = 2
first1 = 0
void RecursiveReverse(node** headRef) {
    node* first;
    node* rest;

    if (*headRef == NULL) return;

    first = *headRef;
    rest = first->next;
    if (rest == NULL) return;
    RecursiveReverse(&rest);

    first->next->next = first;
    first->next = NULL;

    *headRef = rest;
}
void RecursiveReverse(node** headRef) {
    node* first;
    node* rest;

    if (*headRef == NULL) return;

    first = *headRef;
    rest = first->next;
    if (rest == NULL) return;
    RecursiveReverse(&rest);

    first->next->next = first;
    first->next = NULL;

    *headRef = rest;
}
void RecursiveReverse(node** headRef) {
    node* first;
    node* rest;

    if (*headRef == NULL) return;

    first = *headRef;
    rest = first->next;
    if (rest == NULL) return;
    RecursiveReverse(&rest);

    first->next->next = first;
    first->next = NULL;

    *headRef = rest;}

rest₁ = 2
first₁ = 0
void RecursiveReverse(node** headRef) {
    node* first;
    node* rest;

    if (*headRef == NULL) return;

    first = *headRef;
    rest = first->next;
    if (rest == NULL) return;
    RecursiveReverse(&rest);
    first->next->next = first;
    first->next = NULL;
    *headRef = rest;}

Dos and Don’ts
Dos and Don’ts

Don’t lose track of the head

head=head->next;
node* current=head;
    current=current->next;
Dos and Don’ts

Don’t lose track of the head

head=head->next;

node* current=head;
current=current->next;

Do check for NULLs

current=current->next;

while(current!=NULL)
current=current->next;
Dos and Don’ts

Don’t lose track of the head

```c
head = head->next;
```

```
node* current = head;
while (current != NULL)
    current = current->next;
```

Do check for NULLs

```c
current = current->next;
```

```
while (current != NULL)
    current = current->next;
```

Do reassign the tail

```c
tail->next = new_node;
return;
```

```
tail->next = new_node;
new_node->next = NULL;
return;
```
Dos and Don’ts

Don’t lose track of the head

```c
head=head->next;
node* current=head;
current=current->next;
```

Do check for NULLs

```c
current=current->next;
while(current!=NULL)
current=current->next;
```

Do reassign the tail

```c
tail->next=new_node;
return;
tail->next=new_node;
new_node->next=NULL;
return;
```

Do free memory

```c
left->next=to_delete->next;
return;
left->next=to_delete->next;
free(to_delete);
return;
```
Other Dynamic DS
Other Dynamic DS

```c
struct tnode{
    int data;
    struct tnode* left;
    struct tnode* right;
};
```
Other Dynamic DS

```c
struct tnode{
    int data;
    struct tnode* left;
    struct tnode* right;
};
```

```c
struct vertex{
    int data;
    struct edges* edges;
    ...
};
```
B-Tree Animation

https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html