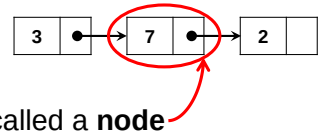
Linked Lists

Lists of Nodes

- Linked lists use pointers to go to the next element



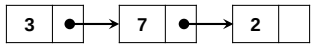
  - each block is called a **node**

Let's implement it:

- a node consists of
  - a data element — an int here
  - a pointer to the next node

```
struct list_node {
  int data;
 struct list_node* next;
};
```

- The whole list is a pointer to its first node

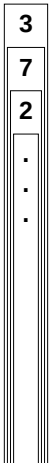


Lists of Nodes

```
struct list_node {
  int data;
 struct list_node* next;
};
```

- Linked lists are a **recursive type**
  - a struct list_node is defined in terms of itself
- What if we don't have this pointer?

  a node that contains an int and
  a node that contains an int and
  a node that contains an int and
  …
  - It would take an *infinite amount of memory!*
  - The C0 compiler disallows this
    - ➢ recursion can only occur behind a pointer (or an array)



Lists of Nodes

```
struct list_node {
  int data;
 struct list_node* next;
};
```

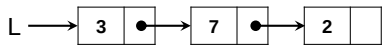
- Let's make it more readable

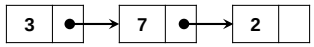
```
typedef struct list_node list;  // ADDED

struct list_node {
  int data;
  list* next;                   // MODIFIED
};
```

This can go before or after the struct

- Implementing this linked list

```
list* L = alloc(list);
L->data = 3;
L->next = alloc(list);
L->next->data = 7;
L->next->next = alloc(list);
L->next->next->data = 2;
```

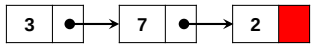




Lists of Nodes

- Does this help us implement queues?
  - Linked lists can be arbitrarily large or small
    - use just the nodes we need
    - size is not fixed like arrays
  - It's easy to insert an element at the beginning
    - allocate a new node and point its next field to the list
  - In fact, it's easy to insert an element between any two nodes
    - allocate a new node and move pointers around
- What about inserting an element at the end?
  - How do we indicate the end of a linked list?

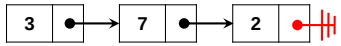
So far we just drew an empty box …



The End of a List

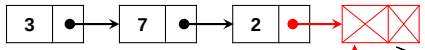
We need to make the pointer in the last node **special**

- Use the NULL pointer



This is a great idea if we don't need direct access to the end of the list

  - This is a **NULL-terminated list**
- Point it to a special node we keep track of somewhere

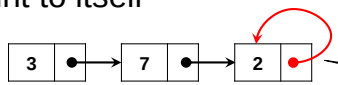


This is a great idea if we **do** need direct access to the end of the list

  - We know we reached the end of the list if its next field is equal to the address of the dummy node

This node is called the **dummy node** or the **sentinel**

- Have it point to itself

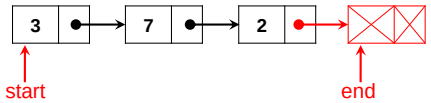


This works too, but nobody does that

List Segments

Lists with a Dummy Node

- We need to keep track of *two* pointers



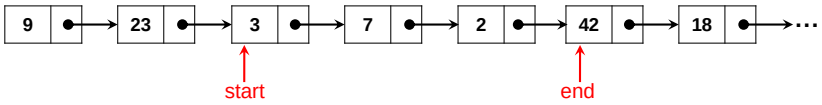
  - **start**: where the first node is
  - **end**: the address in the next field of the last node
    - the address of the dummy node
- What's in the dummy node?
  - some values that are not important to us
    - some number and some pointer
  - we say its fields are *unspecified*
    - no way to test for "unspecified"

These values are not special in any way:
- data could be any element
- next may or may not be NULL

- A dummy value is a value we **don't care** what it is

List Segments

- There may be more nodes before and after



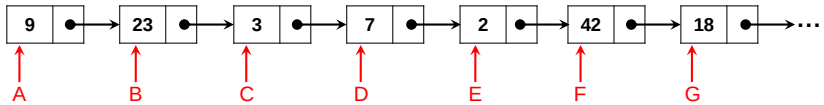
  - The pair of pointers start and end identify our list exactly
    - start is **inclusive** (the first node of the list)
    - end is **exclusive** (one past the last node of the list)

points to the dummy node

  - They identify the **list segment** [start, end)
    - here it contain values 3, 7 and 2
    - similar to array segments A[lo, hi)

List Segments

- There are many list segments in a list



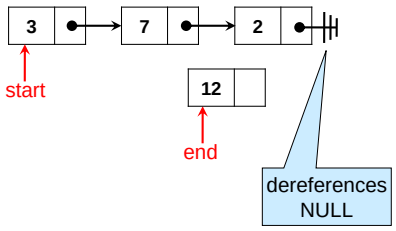
  - The list segment [C, F) contains elements 3, 7, 2
    - its dummy node contains 42 and the pointer G
  - The list segment [A, G) contains 9, 23, 3, 7, 2, 42
    - its dummy node contains 18 and the some pointer
  - The list segment [B, D) contains 23, 3
    - its dummy node contains 7 and the pointer E
  - The list segment [C, C) contains no elements
    - its dummy node contains 3 and the pointer D
    - this is the **empty segment**
    - any segment where start is the same as end
      - [A, A), [B, B), …

Checking for List Segments

```
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};
```

- We want to write a specification function that checks that two pointers start and end form a list segment
  - Follow the next pointer from start until we reach end

```
bool is_segment(list* start, list* end) {
  list* l = start;
  while (l != end) {
    l = l->next;
  }
  return true;
}
```



  - Does this work?
    - the dereference l->next may not be safe
      - we need NULL-checks!
    - we never return false

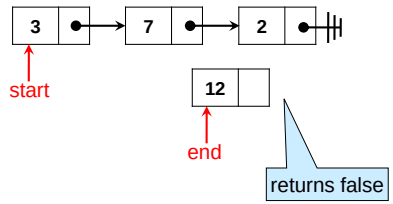
✘

Checking for List Segments

```
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};
```

- *We want to write a specification function that checks that two pointers start and end form a list segment*
  - *Follow the next pointer from start until we reach end*

```
bool is_segment(list* start, list* end) {
  list* l = start;
  while (l != NULL) {          // MODIFIED
    if (l == end) return true; // ADDED
    l = l->next;
  }
  return false;                // MODIFIED
}
```



  - Does this work?
    - if there is a list segment from start to end, it will return true
    - if it returns false, there is no list segment from start to end
  - It works then …

Checking for List Segments

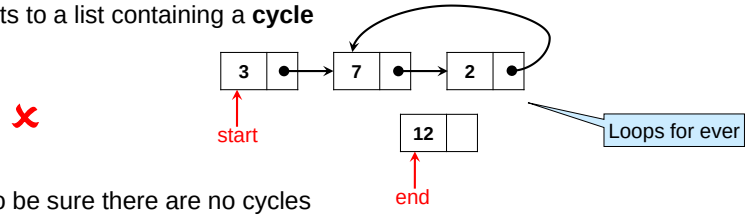
```
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};
```

- A function that checks that start and end form a list segment

```
bool is_segment(list* start, list* end) {
  list* l = start;
  while (l != NULL) {
    if (l == end) return true;
    l = l->next;
  }
  return false;
}
```

- *if there is a list segment from start to end, it will return true*
- *if it returns false, there is no list segment from start to end*

- Can there be no list segment but it does not return false
  - if start points to a list containing a **cycle**



  - We need to be sure there are no cycles

Checking for List Segments

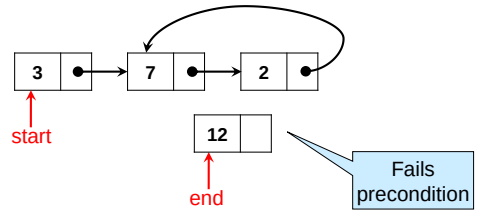
```
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};
```

- A function that checks that start and end form a list segment
  - We need to be sure there are no cycles

```
bool is_segment(list* start, list* end)
//@requires is_acyclic(start);     // ADDED
{
  list* l = start;
  while (l != NULL) {
    if (l == end) return true;
    l = l->next;
  }
  return false;
}
```

We will implement it later

  - Does this work?
    - Yes!



Checking for List Segments

```
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};
```

- A function that checks that start and end form a list segment

```
bool is_segment(list* start, list* end)
//@requires is_acyclic(start);
{
  list* l = start;
  while (l != NULL) {
    if (l == end) return true;
    l = l->next;
  }
  return false;
}
```

  - Notes:
    - returns false if start == NULL
    - or if end == NULL
      - NULL is not a pointer to a list node
      - subsumes NULL-check for both start and end

Checking for List Segments

```
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};
```

- We can also write it more succinctly
  - using a for loop

```
bool is_segment(list* start, list* end)
//@requires is_acyclic(start);
{
  for (list* l = start; l != NULL; l = l->next) {
    if (l == end) return true;
  }
  return false;
}
```

All 3 versions are equivalent

  - recursively

```
bool is_segment(list* start, list* end)
//@requires is_acyclic(start);
{
  if (start == NULL) return false;
  return start == end
         || is_segment(start->next, end);
}
```

Detecting Cycles

- How to check if a list is cyclic?
  - Use a counter and look for overflows
    - very inefficient!
    - also, C0 pointers are 64 bits but ints are 32 bits

✗ In C0, there are more pointers than integers!

  - Keep track of visited nodes somewhere
    - in an array?
    - in another list?

✗ how big to make it? array indices are 32 bits

✗ how do we check it has no cycles?

  - Add a “visited” field to the nodes (a boolean)
    - we need to know the list is acyclic to initialize it to false!

✗

  - What then?

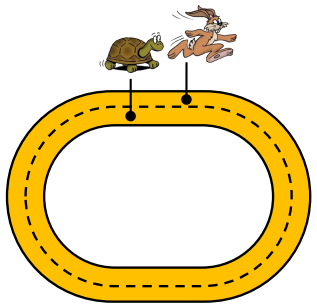
Detecting Cycles

- The tortoise and hare algorithm
  - Traverse the list using two pointers
    - the tortoise starts at the beginning and moves by 1 step
    - the hare starts just ahead of the tortoise and moves by 2 steps
  - If the hare ever overtakes the tortoise, there is a cycle

by this dude

Robert W. Floyd

```
bool is_acyclic(list* start) {
  if (start == NULL) return true;
  list* t = start;              // tortoise
  list* h = start->next;        // hare
  while (h != t) {
    if (h == NULL || h->next == NULL) return true;
    //@assert t != NULL;        // hare hits NULL quicker
    t = t->next;                // tortoise moves by 1 step
    h = h->next->next;          // hare moves by 2 steps
  }
  //@assert h == t;             // hare has overtaken tortoise
  return false;
}
```



Detecting Cycles

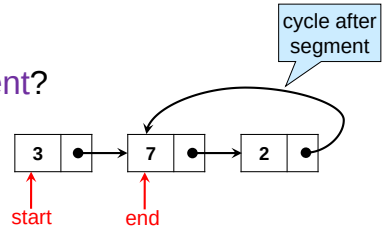
- The tortoise and hare algorithm

```
bool is_acyclic(list* start) {
  if (start == NULL) return true;
  list* t = start;              // tortoise
  list* h = start->next;        // hare
  while (h != t) {
    if (h == NULL || h->next == NULL) return true;
    //@assert t != NULL;        // hare hits NULL quicker
    t = t->next;                // tortoise moves by 1 step
    h = h->next->next;          // hare moves by 2 steps
  }
  //@assert h == t;             // hare has overtaken tortoise
  return false;
}
```

  - Returns
    - true if there is no cycle
    - false if there is a cycle

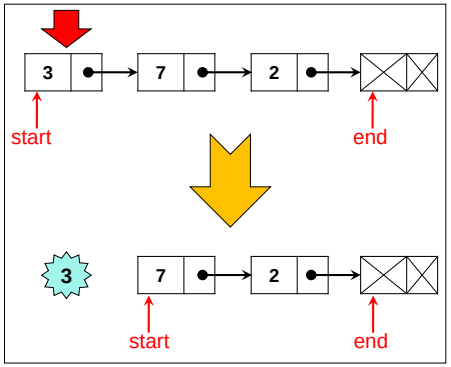
- Does it fix our problem with is_segment?
  - Too aggressive
  - *Exercise: fix it!*

***Hint**: you need to account for end*



Manipulating List Segments

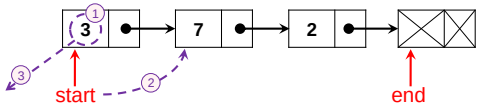
Deleting an Element



- How do we remove the node **at the beginning** of a non-empty list segment [start, end)?
  - ➢ and return the value in there

1. grab the value in the start node
2. move start to point to the next node
3. return the value

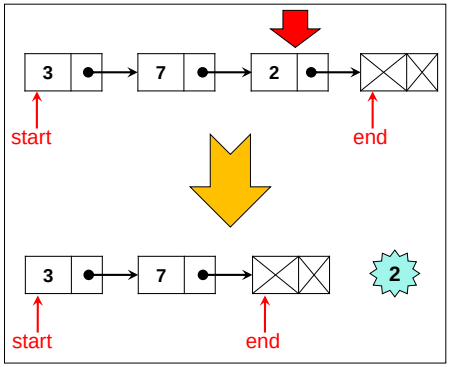
```
1 int x = start->data;
2 start = start->next;
3 return x;
```



- **Complexity**: O(1)

Note: we are not "deleting" the node, just making the segment shorter

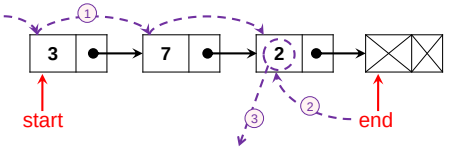
Deleting an Element



- How do we remove the **last** node of a non-empty list segment [start, end)?
  - ➢ and return the value in there
  - ○ we must go from start
    - ❑ end is one node too far

1. follow next until just before end
2. move end to that node
3. return its value

```
1 list* l = start;
  while (l->next != end)
    l = l->next;
2 end = l;
3 return l->data;
```

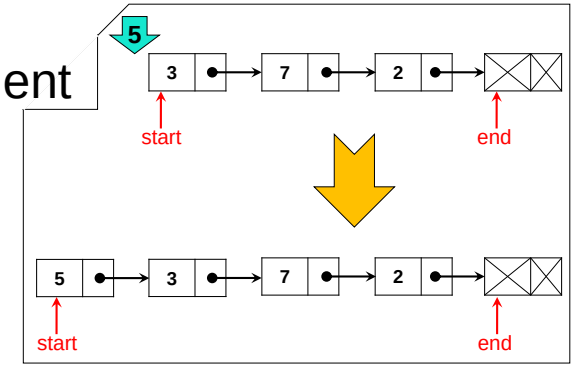


- **Complexity**: O(n)

Expensive!

Notes:
- The old last node becomes the new dummy node
- We are not "deleting" anything, just making the segment shorter

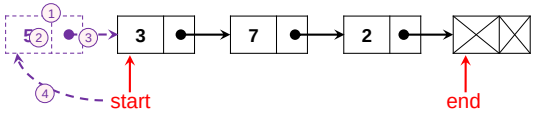
Inserting an Element



- How do we add a node **at the beginning** of a list segment [start, end)?

1. create a new node
2. set its data field to the value to add
3. set its next field to start
4. set start to it

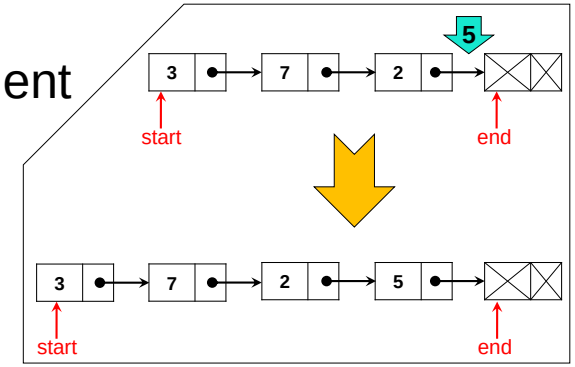
```
1 list* l = alloc(list);
2 l->data = x;
3 l->next = start;
4 start = l;
```



- **Complexity**: O(1)

Note: we are adding a brand new node

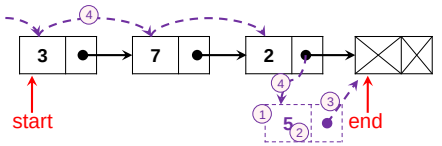
Inserting an Element



- How do we add a node **as the last** node of a list segment [start, end)?

1. create a new node
2. set its data field to the value to add
3. set its next field to end
4. point the old last node to it

```
1 list* new_last = alloc(list);
2 new_last->data = x;
3 new_last->next = end;
  list* l = start;
  while (l->next != end)
    l = l->next;
4 l->next = new_last;
```

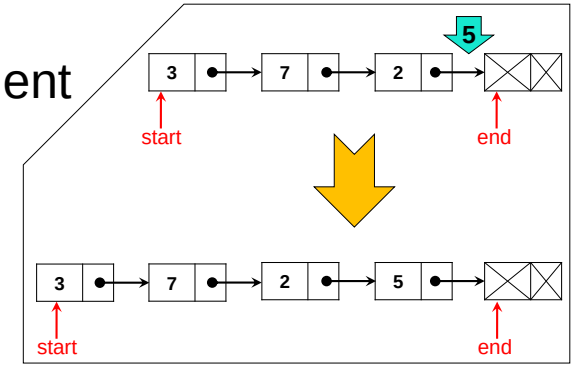


- **Complexity**: O(n)

Expensive!

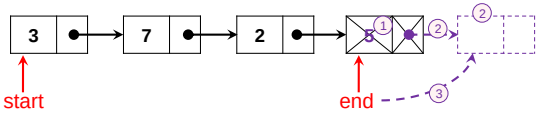
Note: we are adding a new last node, but we modify the next pointer of the old last node

Inserting an Element



- How do we add a node **as the last** node of a list segment [start, end)?
  - ***Can we do better?***
    1. set the data field of end to the value to add
    2. set its next field to a new dummy node
    3. set end to it

```
① end->data = x;
② end->next = alloc(list);
③ end = end->next;
```



  - **Complexity**: O(1)

Much better!

Note: we are using the old dummy node as the new last node, and creating a new dummy

Summary

	at the beginning	*at the end*
Inserting	O(1)	O(1)
Deleting	O(1)	O(n)

Good

Bad

- We will use this as a guide when implementing queues (and stacks) to achieve their complexity goals