CSCI 104
Linked Lists

Mark Redekopp

Revised: 01/2022

Lists

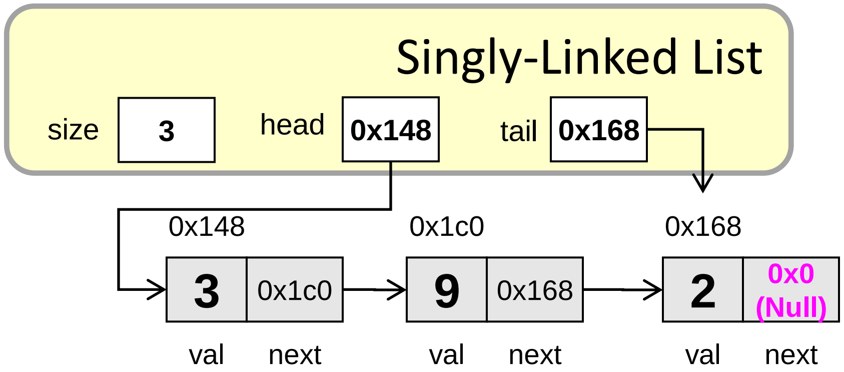
- Ordered collection of items, which may contain duplicate values, usually accessed based on their position (index)
  - Ordered = Each item has an index and there is a front and back (start and end)
  - Duplicates allowed (i.e. in a list of integers, the value 0 could appear multiple times)
  - Accessed based on their position (list[0], list[1], etc.)
- What are the operations you perform on a list?

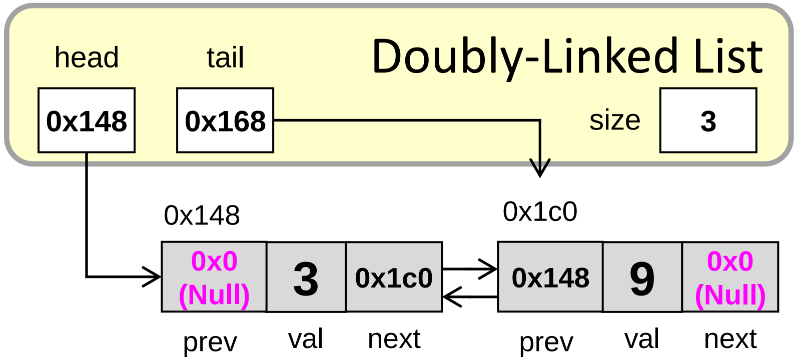
List Operations

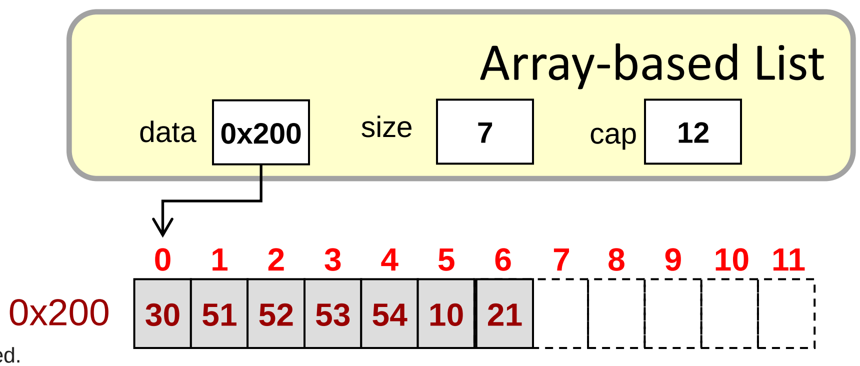
Operation	Description	Input(s)	Output(s)
insert	Add a new value at a particular location shifting others back	Index : int Value	
remove	Remove value at the given location	Index : int	Value at location
get / at	Get value at given location	Index : int	Value at location
set	Changes the value at a given location	Index : int Value	
empty	Returns true if there are no values in the list		bool
size	Returns the number of values in the list		int
push_back / append	Add a new value to the end of the list	Value	
find	Return the location of a given value	Value	Int : Index

List Implementation Options

- Singly-Linked List
  - With or without tail pointer
- Doubly-Linked List
  - With or without tail pointer
- Array-based List







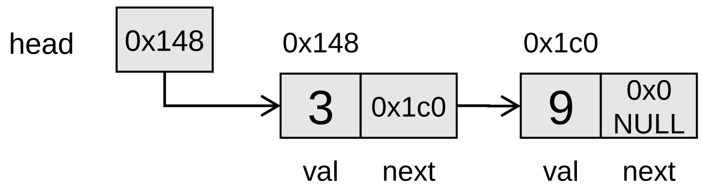
Implementation Options

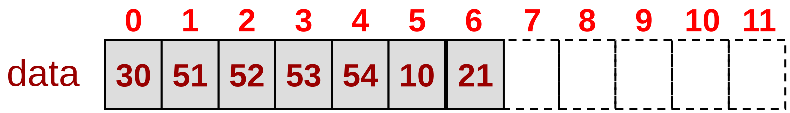
Linked Implementations

- Allocate each item separately
- Random access (get the i-th element) is O(___)
- Adding new items never requires others to move
- Memory overhead due to pointers

Array-based Implementations

- Allocate a block of memory to hold many items
- Random access (get the i-th element) is O(___)
- Adding new items may require others to shift positions
- Memory overhead due to potentially larger block of memory with unused locations





LINKED IMPLEMENTATIONS

Note

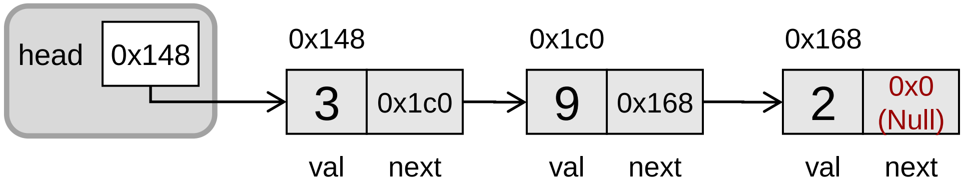
- The basics of linked list implementations was taught in CS 103
  - We assume that you already have basic exposure and practice using a class to implement a linked list
  - We will highlight some of the more important concepts

Linked List

- Use structures/classes and pointers to make ‘linked’ data structures
- A linked list is…
  - Arbitrarily sized collection of values
  - Can add any number of new values via **dynamic memory allocation**
  - Supports typical List ADT operations:
    - Insert
    - Get
    - Remove
    - Size (*Should we keep a size data member?*)
    - Empty
- Can define a List class to encapsulate the head pointer and operations on the list

```
#include<iostream>
using namespace std;

struct Item {
  int val;
  Item* next;
};

class List
{
  public:
   List();
   ~List();
   void push_back(int v); ...
  private:
   Item* head_;
};
```

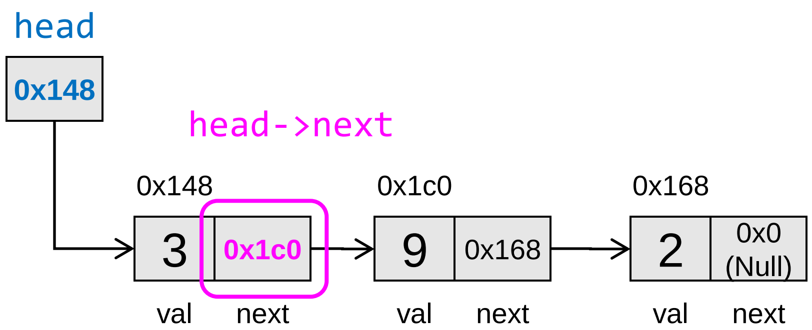
Item blueprint:



Rule of thumb: Still use ‘structs’ for objects that are purely collections of data and don’t really have operations associated with them. Use ‘classes’ when data does have associated functions/methods.

A Common Misconception

- Important Note:
  - 'head' is NOT an Item, it is a pointer to the first item
  - Sometimes folks get confused and think head is an item and so to get the location of the first item they write 'head->next'
  - In fact, head->next evaluates to the 2^{nd} items address



head->next yields a pointer to the 2^{nd} item!
head yields a pointer to the 1^{st} item!

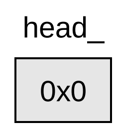
Don't Need Classes

- Notice the class on the previous slide had *only 1 data member* (the head pointer)
- We don't have to use classes…
  - The class just acts as a wrapper around the head pointer and the operations
  - So while a class is probably the correct way to go in terms of organizing your code, for today we can show you a less modular, procedural approach
- Define functions for each operation and pass it the head pointer as an argument

```
#include<iostream>
using namespace std;
struct Item {
  int val;
  Item* next;
};
// Function prototypes
void append(Item*& head, int v);
bool empty(Item* head);
int size(Item* head);

int main()
{
  Item* head1 = NULL;
  Item* head2 = NULL;
  int size1 = size(head1);
  bool empty2 = empty(head2);
  append(head1, 4);
}
```

Item blueprint:

class List:

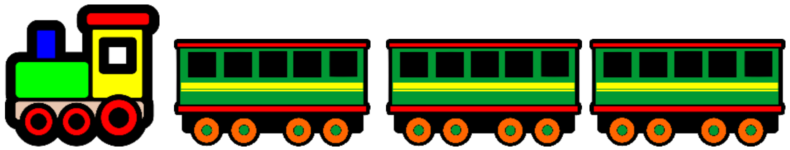


Rule of thumb: Still use 'structs' for objects that are purely collections of data and don't really have operations associated with them. Use 'classes' when data does have associated functions/methods.

Linked List Implementation

- To maintain a linked list you need only keep one data value: head
  - Like a train engine, we can attach any number of 'cars' to the engine
  - The engine looks different than all the others
    - In our linked list it's just a single pointer to an Item
    - All the cars are Item structs
    - Each car has a hitch for a following car (i.e. next pointer)

```
#include<iostream>

struct Item {
  int val;
  Item* next;
};

void append(Item*& head, int v);

int main()
{
  Item* head1 = NULL;
  Item* head2 = NULL;

}
```



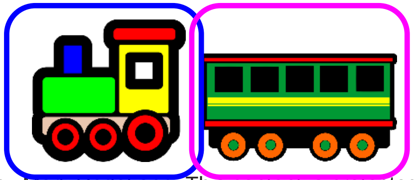
Engine = "head"

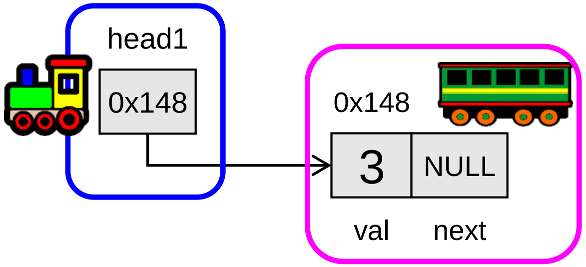
Each car = "Item"

Append

- Adding an item (train car) to the back can be split into 2 cases:
  - Case 1: Attaching the car to the engine (i.e. the list is empty and we have to change the head pointer)
    - Changing the head pointer is a special case since we must ensure that change propagates to the caller
  - Case 2: Attaching the car to another car (i.e. the list has other Items already) and so we update the next pointer of an Item

```
#include<iostream>
using namespace std;
struct Item {
  int val;
  Item* next;
};

void append(Item*& head, int v)
{
  if(head == NULL){
    head = new Item;
    head->val = v; head->next = NULL;
  }
  else {...}
}

int main()
{
  Item* head1 = NULL;
  Item* head2 = NULL;
  append(head1, 3);
}
```

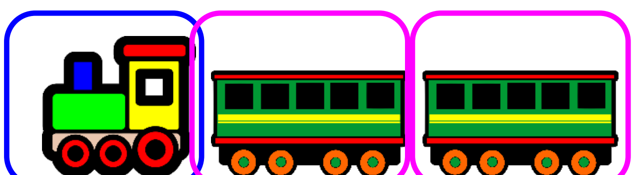


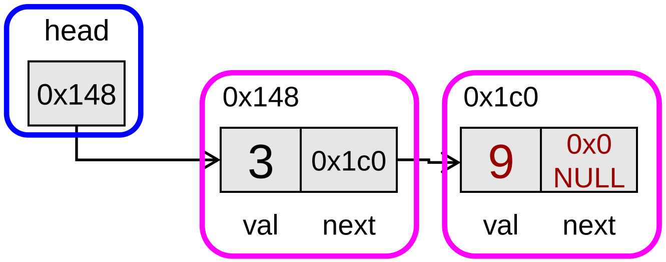


Linked List

- Adding an item (train car) to the back can be split into 2 cases:
  - Attaching the car to the engine (i.e. the list is empty and we have to change the head pointer)
  - Attaching the car to another car (i.e. the list has other Items already) and so we update the next pointer of an Item

```
#include<iostream>
using namespace std;
struct Item {
  int val;
  Item* next;
};

void append(Item*& head, int v)
{
  if(head == NULL){
    head = new Item;
    head->val = v; head->next = NULL;
  }
  else {...}
}

int main()
{
  Item* head1 = NULL;
  Item* head2 = NULL;
  append(head1,3); append(head1,9);
}
```



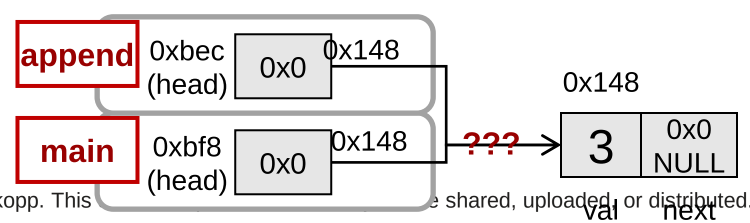


Passing Pointers "by-Value"

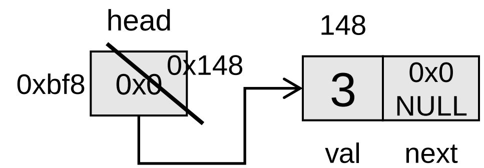
- Look at how the head parameter is passed…Can you explain it?
  - Append() may need to change the value of head and we want that change to be visible back in the caller.
  - Even pointers are passed by value…wait, huh?
  - When one function calls another and passes a pointer, it is the data being pointed to that can be changed by the function and seen by the caller, but the pointer itself is passed by value.
  - You email your friend a URL to a Google doc. The URL is copied when the email is sent but the document being referenced is shared.
  - If we want the pointer to be changed and visible we need to pass the pointer by reference
  - We choose Item*& but we could also pass an Item**

```
void append(Item*& head, int v)
{
  Item* newptr = new Item;
  newptr->val = v; newptr->next = NULL;

  if(head == NULL){
    head = newptr;
  }
  else {
    Item* temp = head;
    // iterate to the end
    ...
  }
}
```

```
void append(Item** head, int v)
{
  Item* newptr = new Item;
  newptr->val = v; newptr->next = NULL;

  if(*head == NULL){
    *head = newptr;
  }
  else {
    Item* temp = *head;
    // iterate to the end
    ...
  }
}
```



Passing Pointers by…

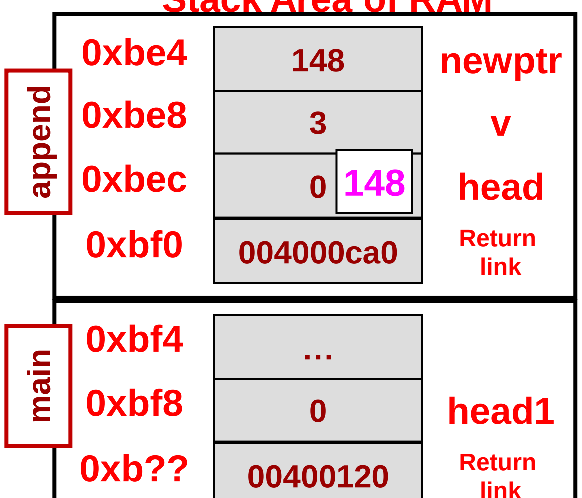


Pointer Passed-by-Value

```
int main() {
  Item* head1 = 0;
  append(head1, 3);
```

```
void append(Item* head, int v)
{
 Item* newptr = new Item;
 newptr->val = v;
 newptr->next = NULL;
 if(head == 0){ head = newptr;}
 else {
   Item* temp = head;
    ...
} }
```

Stack Area of RAM

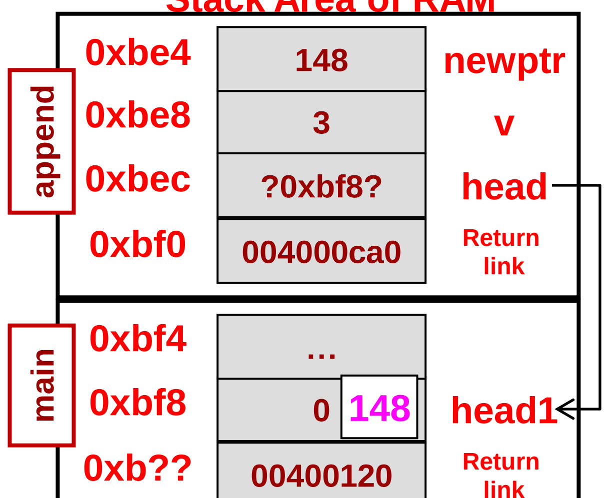


Pointer Passed-by-C++ Reference

```
int main() {
  Item* head1 = 0;
  append(head1, 3);
```

```
void append(Item*& head, int v)
{
 Item* newptr = new Item;
 newptr->val = v;
 newptr->next = NULL;
 if(head == 0){ head = newptr;}
 else {
   Item* temp = head;
    ...
} }
```

Stack Area of RAM

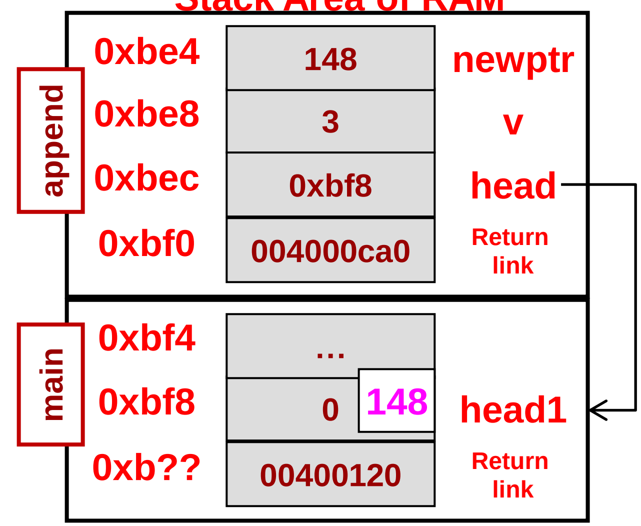


Pointer Passed-by-Pointer Reference

```
int main() {
  Item* head1 = 0;
  append(&head1, 3);
```

```
void append(Item** head, int v)
{
 Item* newptr = new Item;
 newptr->val = v;
 newptr->next = NULL;
 if(*head == 0){ *head = newptr;}
 else {
   Item* temp = *head;
    ...
} }
```

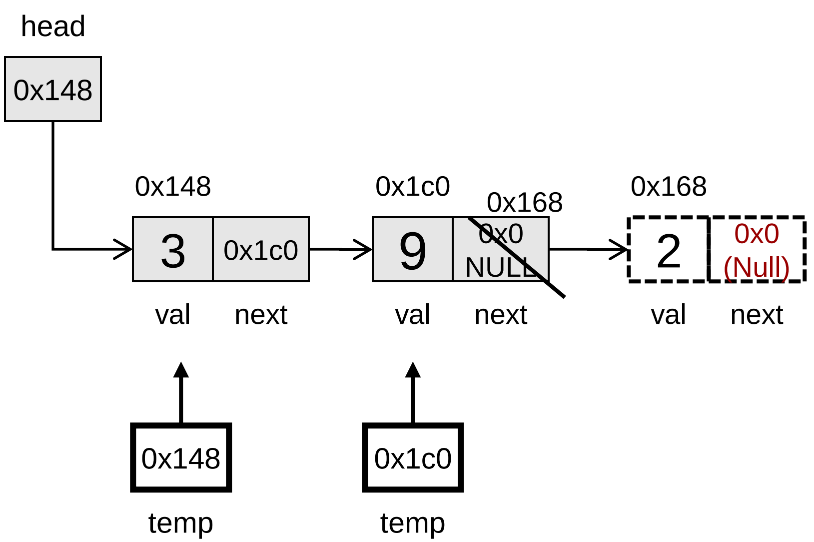
Stack Area of RAM



Iterating Through a Linked List

- Start from head and iterate to end of list
  - Copy head to a temp pointer (because if we modify head we can never recover where the list started)
  - Use temp pointer to iterate through the list until we find the tail (element with next field = NULL)
  - To take a step we use the line: `temp = temp->next;`
  - Optional: Update old tail item to point at new tail item)

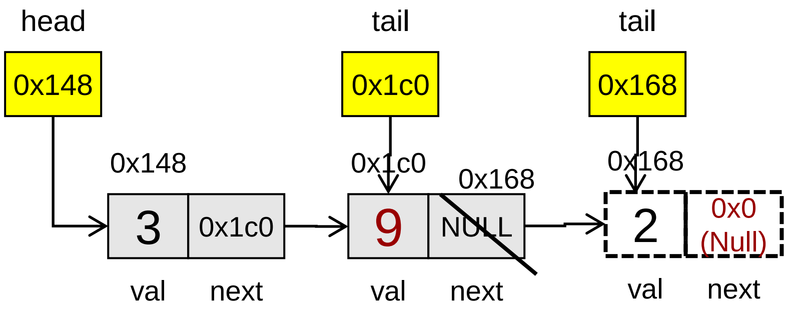
```
void append(Item*& head, int v)
{
  Item* newptr = new Item;
  newptr->val = v; newptr->next = NULL;

  if(head == NULL){
    head = newptr;
  }
  else {
    Item* temp = head;
    // iterate to the end
    ...
  }
}
```



Given only head, we don't know where the list ends so we have to traverse to find it

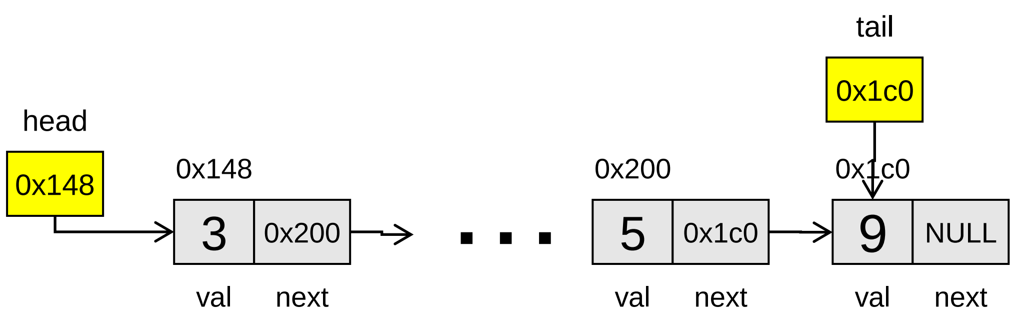
Adding a Tail Pointer

- If in addition to maintaining a head pointer we can also maintain a tail pointer
- A tail pointer saves us from iterating to the end to add a new item
- Need to update the tail pointer when…
  - We add an item to the end
    - Easy, fast!
  - We remove an item from the end
    - _______________________



Removal

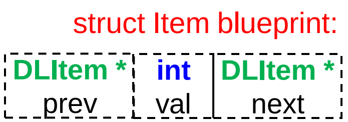
- To remove the last item, we need to update the 2nd to last item (set it's next pointer to NULL)
- We also need to update the tail pointer
- But this would require us to traverse the full list requiring O(n) time
- ONE SOLUTION: doubly-linked list

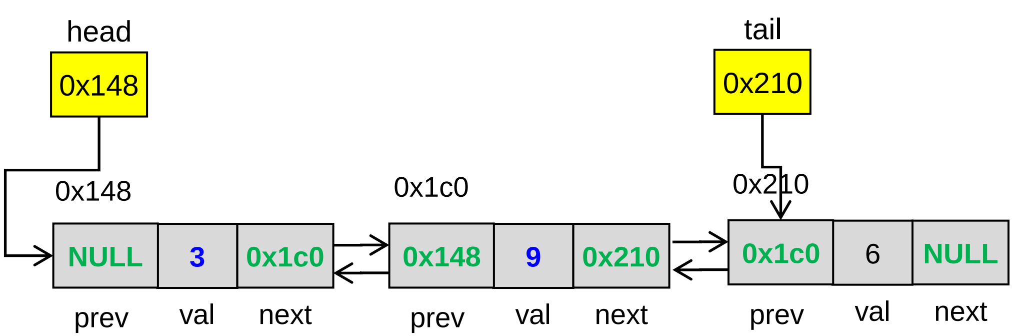


Doubly-Linked Lists

- Includes a previous pointer in each item so that we can traverse/iterate backwards or forward
- First item's previous field should be NULL
- Last item's next field should be NULL
- The key to performing operations is updating all the appropriate pointers correctly!
  - Let's practice identifying this.
  - We recommend drawing a picture of a sample data structure before coding each operation

```
#include<iostream>

using namespace std;
struct DLItem {
  int val;
  DLItem* prev;
  DLItem* next;
};

int main()
{
  DLItem* head, *tail;
};
```





Summary of Linked List Implementations

Operation vs Implementation for Edges	Push_front	Pop_front	Push_back	Pop_back	Memory Overhead Per Item
Singly linked-list w/ head ptr ONLY					1 pointer (next)
Singly linked-list w/ head and tail ptr					1 pointer (next)
Doubly linked-list w/ head and tail ptr					2 pointers (prev + next)

- What is worst-case runtime of get(i)?
- What is worst-case runtime of insert(i, value)?
- What is worst-case runtime of remove(i)?

Key Ideas for Linked Lists

- A head pointer is all that is needed to maintain a linked list
- When iterating…
  - Don't lose the head
  - Given a pointer to an item, taking a step to the next node is accomplished with ptr = ptr->next
  - Carefully consider when to stop: at the end, one before the end, on the desired item, one before the desired item based on what needs to be updated
- For a singly linked list, use of a tail pointer allows for fast insertion at the end but not removal
- When writing functions that take (head) pointers to linked lists:
  - Always ensure you check and handle if the pointer is NULL
  - If the head/pointer will change, consider how to return that new value (or use pass-by-reference)

Summary of Linked List Implementations

Operation vs Implementation for Edges	Push_front	Pop_front	Push_back	Pop_back	Memory Overhead Per Item
Singly linked-list w/ head ptr ONLY	Θ(1)	Θ(1)	Θ(n)	Θ(n)	1 pointer (next)
Singly linked-list w/ head and tail ptr	Θ(1)	Θ(1)	Θ(1)	Θ(n)	1 pointer (next)
Doubly linked-list w/ head and tail ptr	Θ(1)	Θ(1)	Θ(1)	Θ(1)	2 pointers (prev + next)

- What is worst-case runtime of get(i)? Θ(i)
- What is worst-case runtime of insert(i, value)? Θ(i)
- What is worst-case runtime of remove(i)? Θ(i)