CSCI 104
List Implementations

Mark Redekopp
David Kempe
Sandra Batista

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

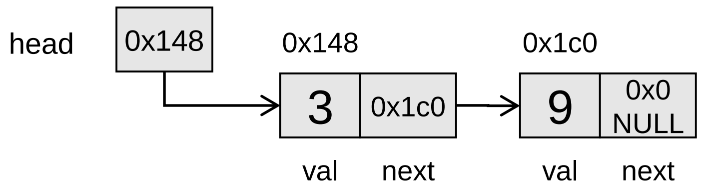
Implementation Options

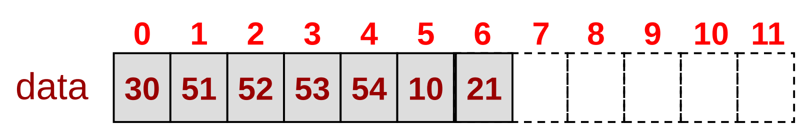
Linked Implementations

- 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

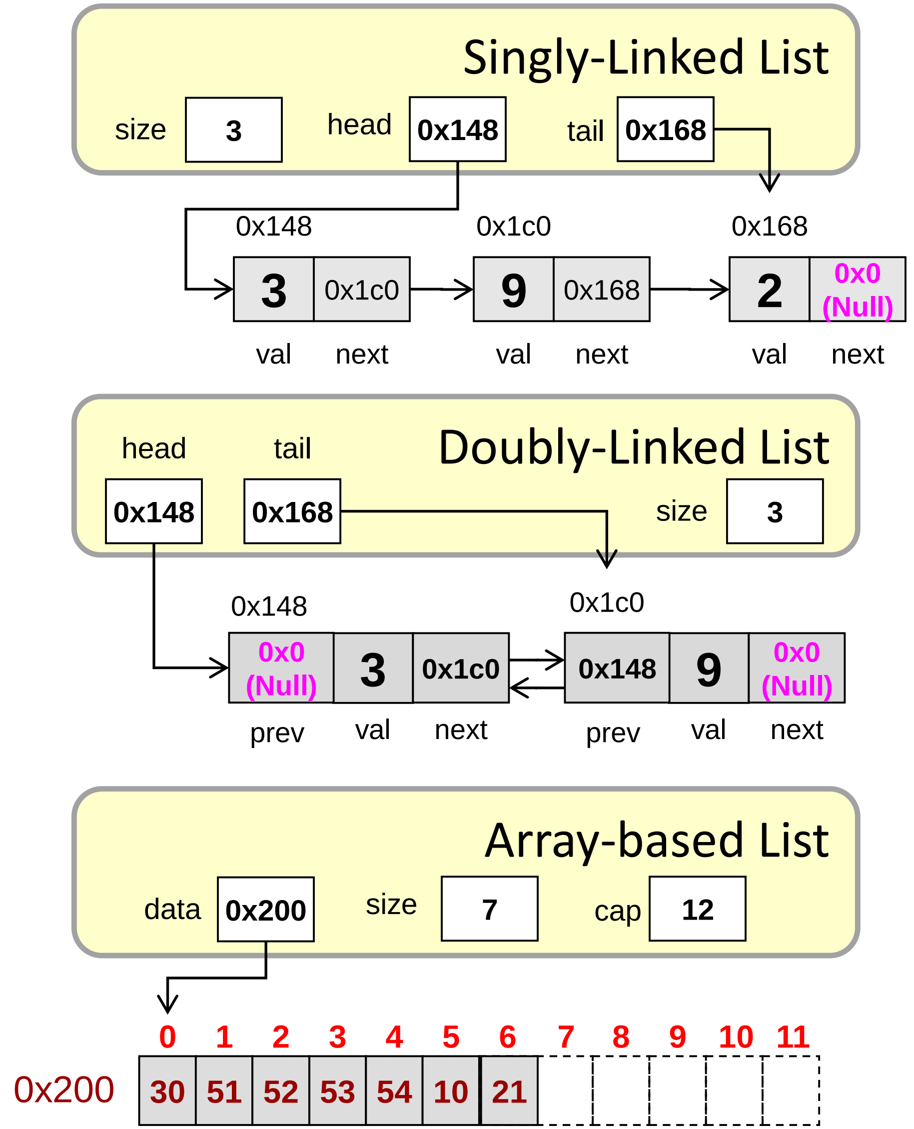
- Allocate a block of memory to hold many items
- Random access (get the i-th element) is O(___)
- Adding new items may require others to shift positions
- Memory overhead due to potentially larger block of memory with unused locations





Implementation Options

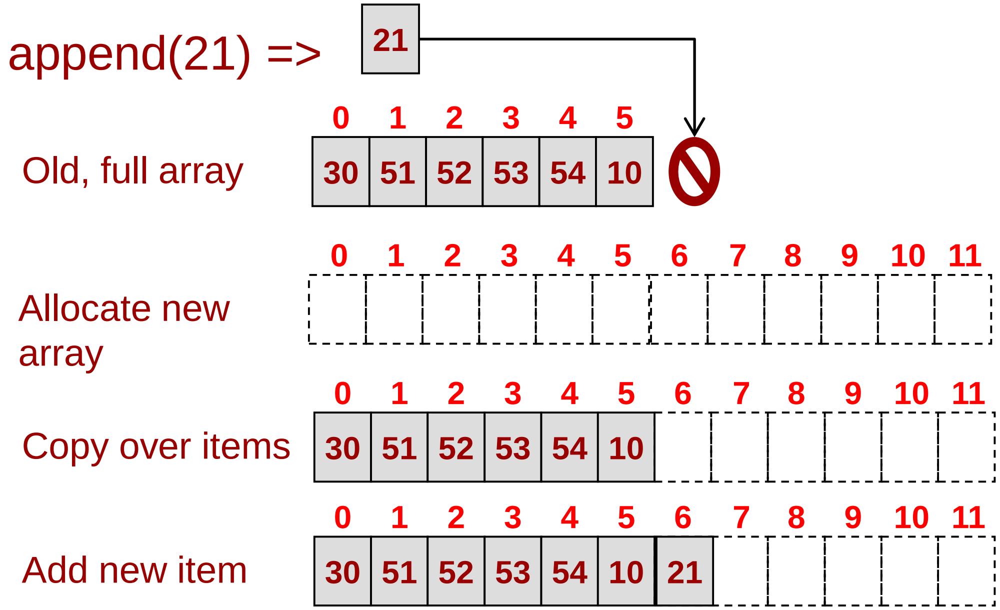
- Singly-Linked List
  - With or without tail pointer
- Doubly-Linked List
  - With or without tail pointer
- Array-based List



LINKED IMPLEMENTATIONS

Array Problems

- Once allocated an array cannot grow or shrink
- If we don't know how many items will be added we could just allocate an array larger than we need but…
  - We might waste space
  - What if we end up needing more…would need to allocate a new array and copy items
- Arrays can't grow with the needs of the client



Motivation for Linked Lists

- Can we create a list implementation that can easily grow or shrink based on the number of items currently in the list
- Observation: Arrays are allocated and deallocated in LARGE chunks
  - It would be great if we could allocate/deallocate at a finer granularity
- Linked lists take the approach of allocating in small chunks (usually enough memory to hold one item)

Bulk Item
(i.e. array)

Single Item
(i.e. linked list)

Note

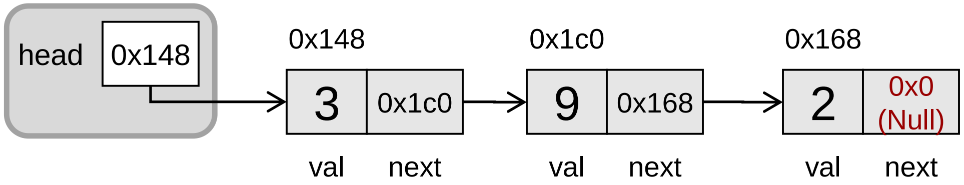
- The basics of linked list implementations was taught in CS 103
  - We assume that you already have basic exposure and practice using a class to implement a linked list
  - We will highlight some of the more important concepts

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.

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:

int	Item*
val	next

class List:

head_

0x0

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.

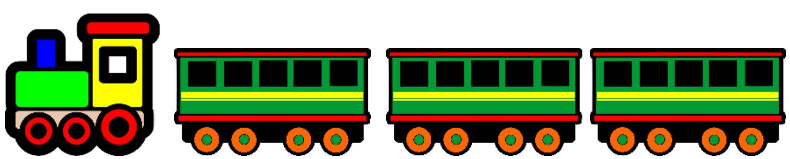
Linked List Implementation

- 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;

}
```

head1

0x0 NULL



Engine = "head"

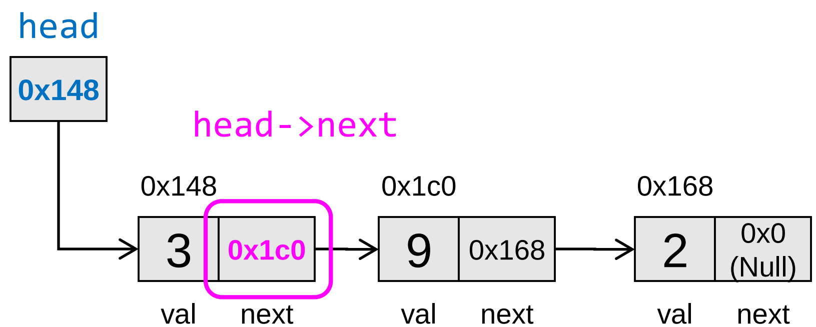
Each car = "Item"

0x0 NULL

head2

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 2nd items address

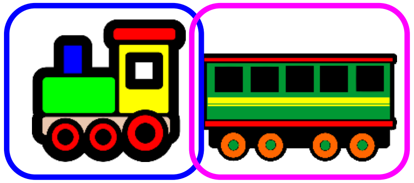


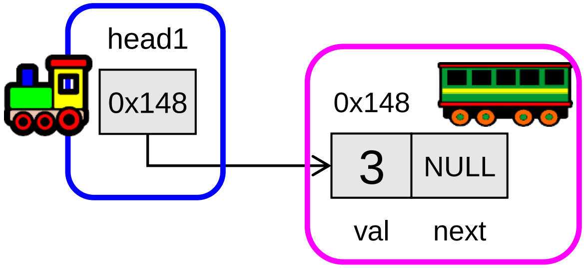
`head->next` yields a pointer to the 2nd item!
`head` yields a pointer to the 1st item!

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);
}
```

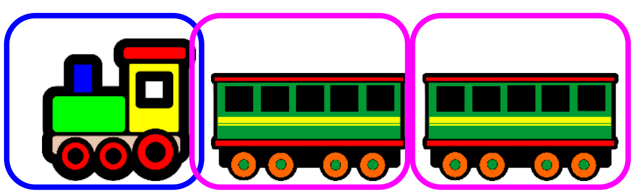


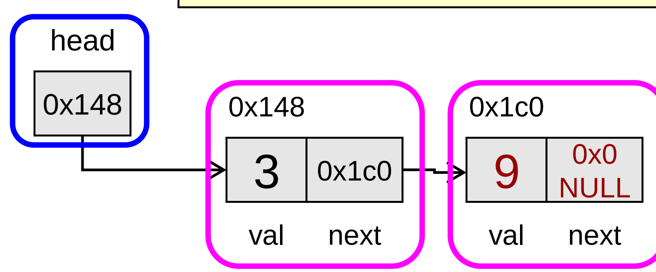


Linked List

- 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);
}
```

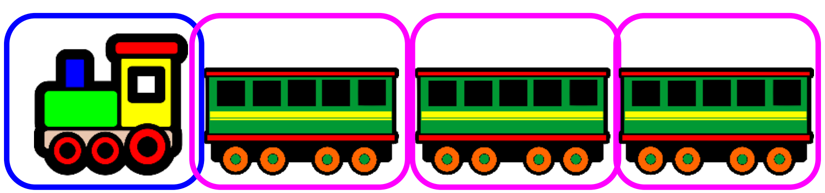


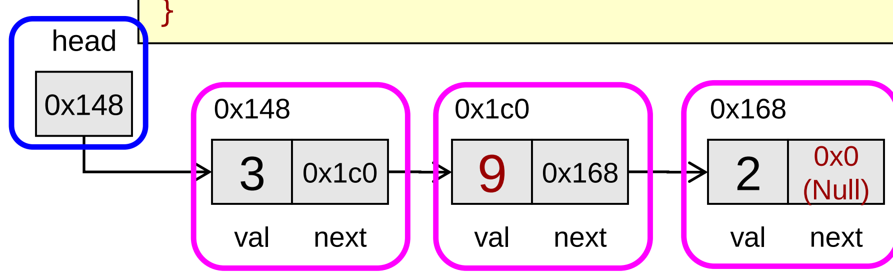


Linked List

- 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);
  append(head1, 2);
}
```

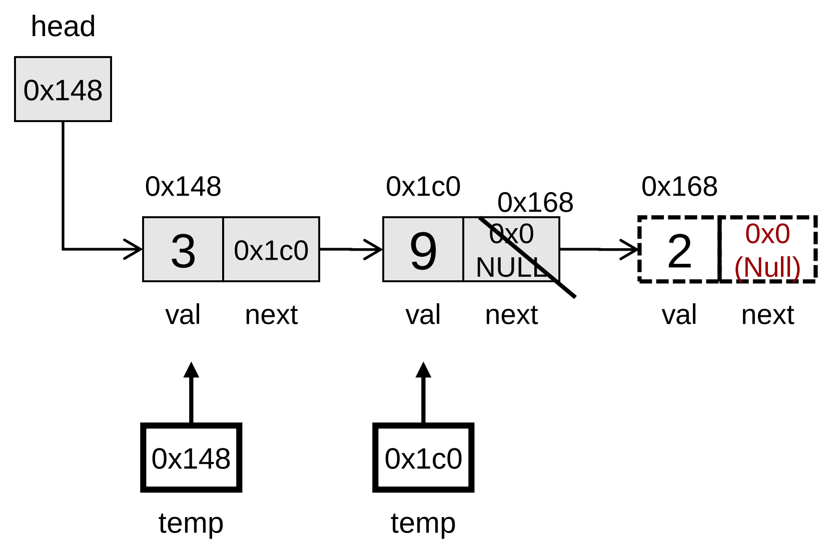




Iterating Through a Linked List

- Start from head and iterate to end of list
  - Allocate new item and fill it in
  - 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;`
  - Update old tail item to point at new tail item

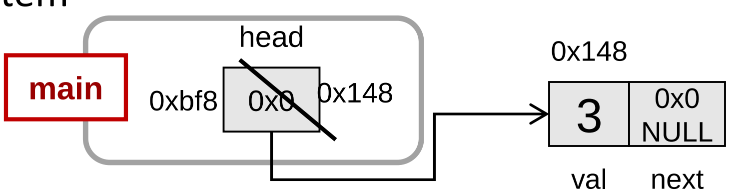
```
void append(Item*& head, int v)
{
  Item* newptr = new Item;
  newptr->val = v; newptr->next = NULL;

  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

Passing Pointers "by-Value"

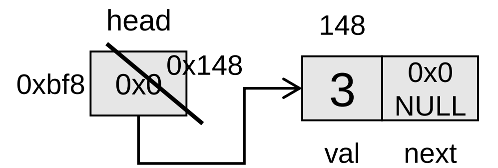
- Look at how the head parameter is passed…Can you explain it?
  - Append() may need to change the value of head and we want that change to be visible back in the caller.
  - Even pointers are passed by value…wait, huh?
  - When one function calls another and passes a pointer, it is the data being pointed 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

	Address	Value	Name
append	0xbe4	148	newptr
	0xbe8	3	v
	0xbec	0 148	head
	0xbf0	004000ca0	Return link
main	0xbf4	...	
	0xbf8	0	head1
	0xb??	00400120	Return link

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

	Address	Value	Name
append	0xbe4	148	newptr
	0xbe8	3	v
	0xbec	?0xbf8?	head
	0xbf0	004000ca0	Return link
main	0xbf4	...	
	0xbf8	0 148	head1
	0xb??	00400120	Return link

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

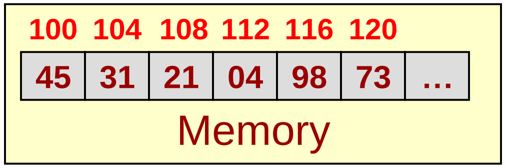
	Address	Value	Name
append	0xbe4	148	newptr
	0xbe8	3	v
	0xbec	0xbf8	head
	0xbf0	004000ca0	Return link
main	0xbf4	...	
	0xbf8	0 148	head1
	0xb??	00400120	Return link

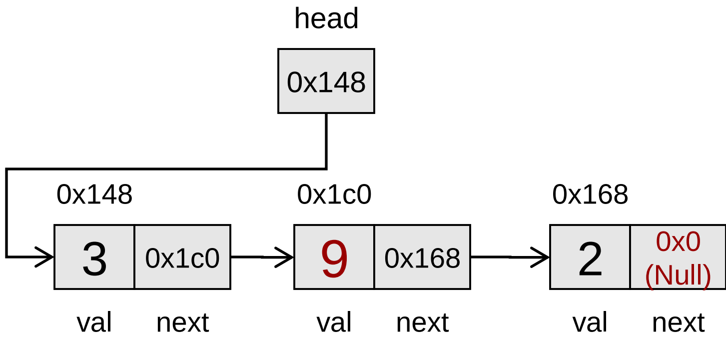
Arrays/Linked List Efficiency

- Arrays are contiguous pieces of memory
- To find a single value, computer only needs
  - The start address
    - Remember the name of the array evaluates to the starting address (e.g. data = 120)
  - Which element we want
    - Provided as an index (e.g. [20])
  - This is all thanks to the fact that items are contiguous in memory
- Linked list items are not contiguous
  - Thus, linked lists have an explicit field to indicate where the next item is
  - This is "overhead" in terms of memory usage
  - Requires iteration to find an item or move to the end

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

int main()
{
  int data[25];
  data[20] = 7;
  return 0;
}
```

data = 100





Using a 'for' Loop to Iterate

- Just as a note, you can use a for loop structure to iterate through a linked list
- Identify the three parts:
  - Initialization
  - Condition check
  - Update statement

```
void print(Item* head)
{
    Item* temp = head;     // init
    while(temp->next){     // condition
      cout << temp->val << endl;
      temp = temp->next;   // update
    }
}
```

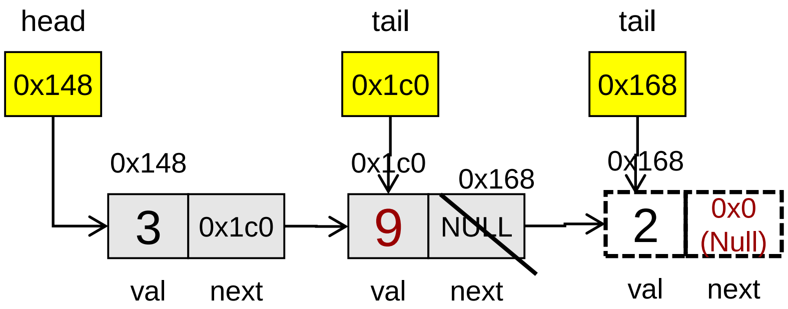
```
void print(Item* head)
{
   for(Item* temp = head;    // init
        temp->next;          // condition
        temp = temp->next)   // update
   {
     cout << temp->val << endl;
   }
}
```

Note: The condition `(temp->next)` is equivalent to `(temp->next != NULL)`. Why?

INCREASING EFFICIENCY OF OPERATIONS + DOUBLY LINKED LISTS

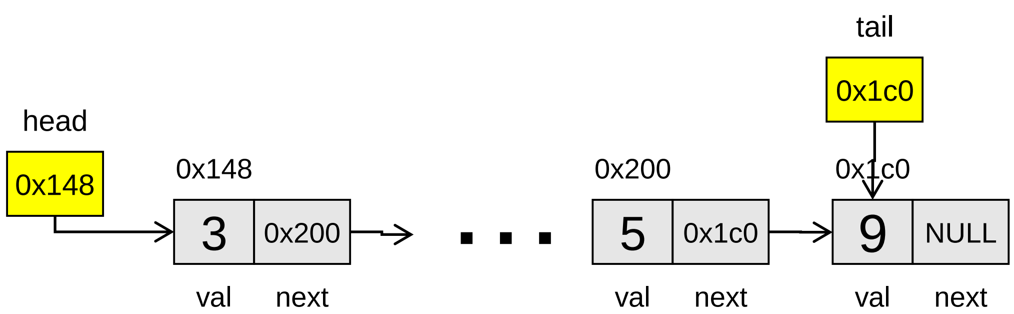
Adding a Tail Pointer

- 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

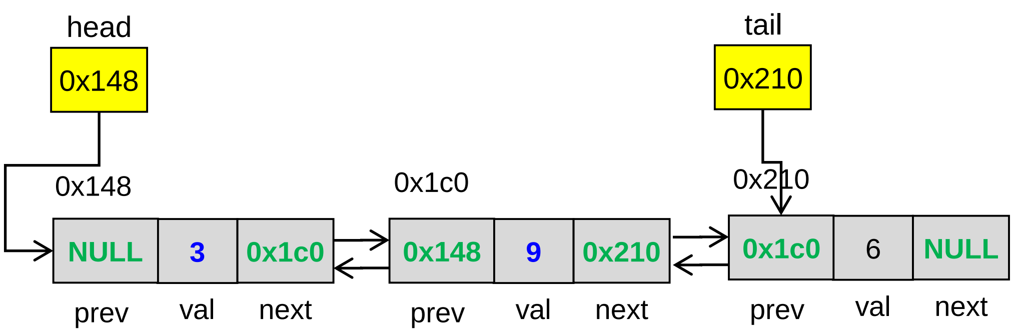


Doubly-Linked Lists

- 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

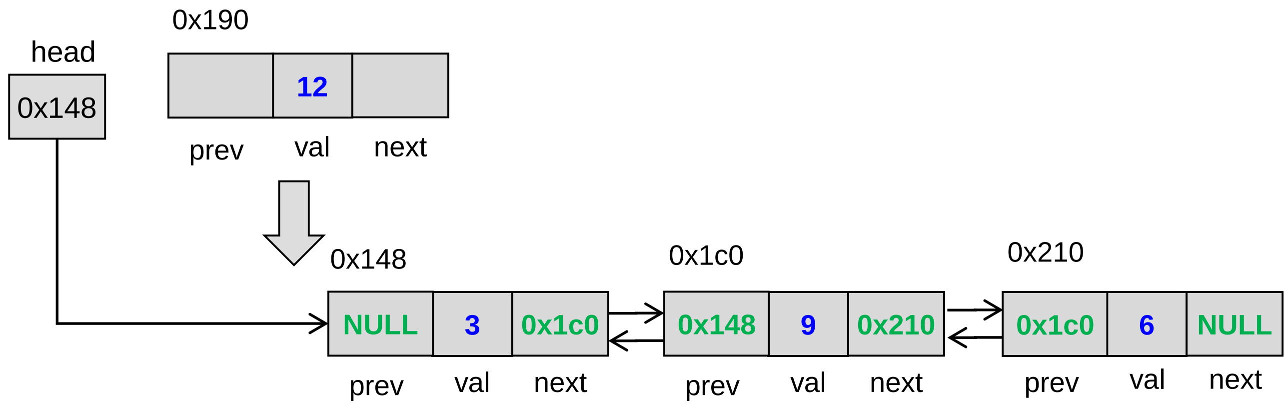
```
#include<iostream>

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

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

struct Item blueprint:



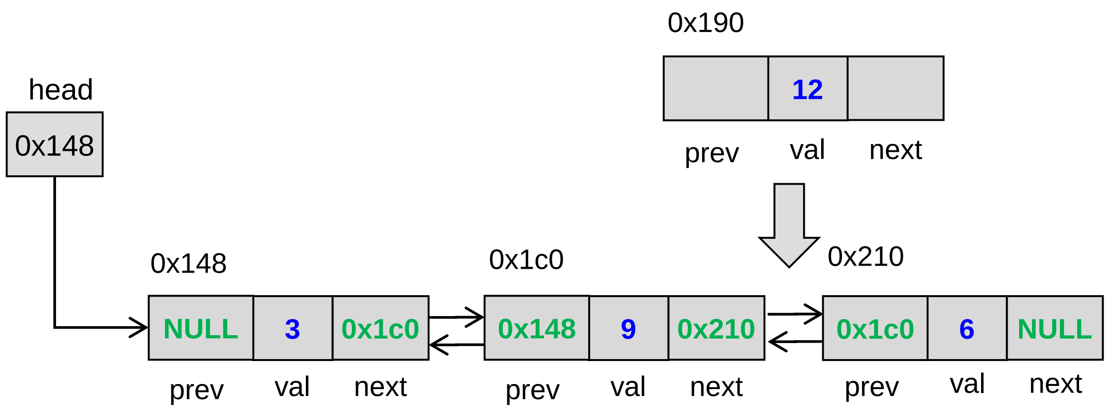
Doubly-Linked List Add Front

- Adding to the front requires you to update…
- …Answer
  - Head
  - New front's next & previous
  - Old front's previous



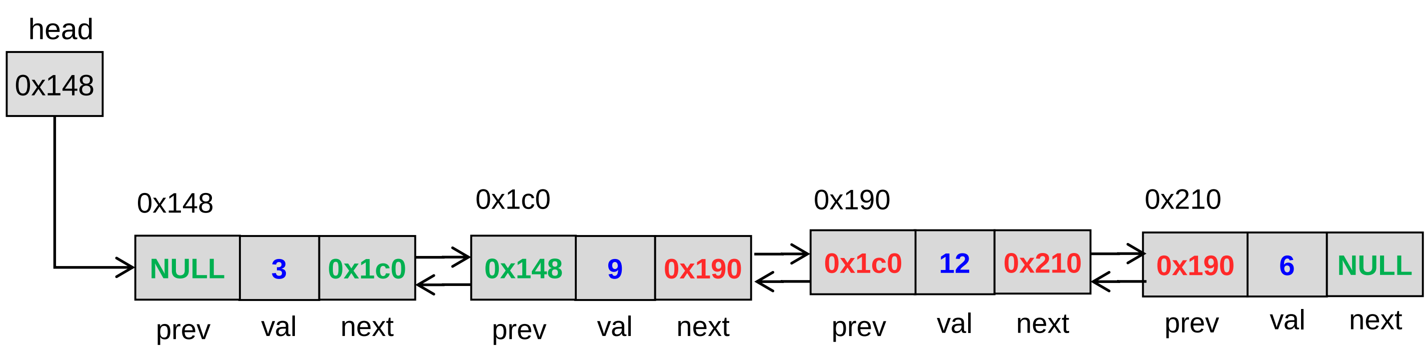
Doubly-Linked List Add Middle

- Adding to the middle requires you to update...
  - Previous item's next field
  - Next item's previous field
  - New item's next field
  - New item's previous field



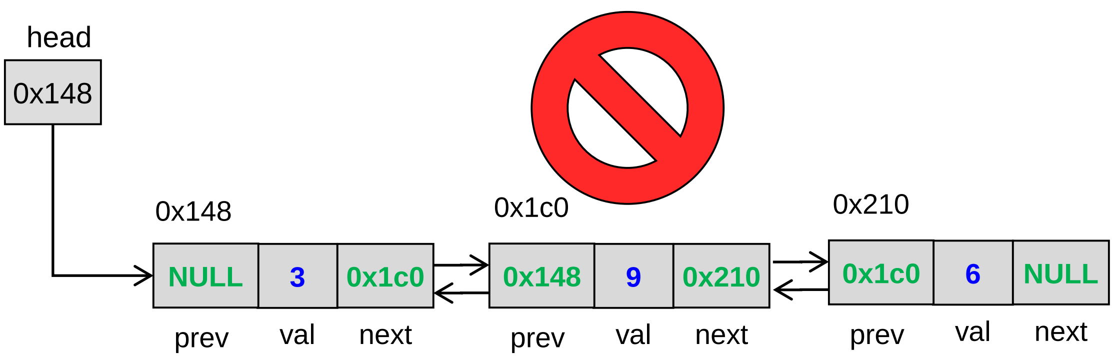
Doubly-Linked List Add Middle

- Adding to the middle requires you to update…
  - Previous item's next field
  - Next item's previous field
  - New item's next field
  - New item's previous field



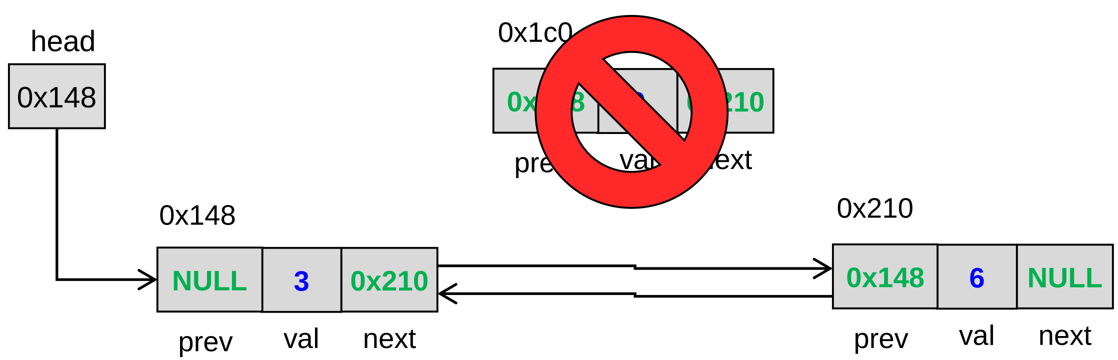
Doubly-Linked List Remove Middle

- Removing from the middle requires you to update…
  - Previous item's next field
  - Next item's previous field
  - Delete the item object



Doubly-Linked List Remove Middle

- Removing from the middle requires you to update…
  - Previous item's next field
  - Next item's previous field
  - Delete the item object



Doubly-Linked List Prepend

- Assume DLItem constructor:
  - `DLItem(int val, DLItem* next, DLItem* prev)`
- Add a new item to front of doubly linked list given head and new value

```
void prepend(DLItem *& head, int n)
{
  DLItem* elem = new DLItem(n, head, NULL);
  head = elem;
  if (head->next != NULL){
    head->next->prev = head;
  }
};
```

Doubly-Linked List Remove

- Remove item given its pointer

```
void remove(DLItem *& head, DLItem *splice)
{
  if (splice != head){
    _____________________________________
  }
  else {
    head = ___________________________;
  }
  if (splice->next != NULL){
    ____________________________________________;
  }
  delete splice;
}
```

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)?

ARRAY-BASED IMPLEMENTATIONS

BOUNDED DYNAMIC ARRAY STRATEGY

A Bounded Dynamic Array Strategy

- Allocate an array of some user-provided size
  - Capacity is then fixed
- What data members do I need?
- Together, think through the implications of each operation when using a bounded array (what issues could be caused due to it being bounded)?

```
#ifndef BALISTINT_H
#define BALISTINT_H

class BAListInt {
 public:
  BAListInt(unsigned int cap);

  bool empty() const;
  unsigned int  size() const;
  void insert(int pos,
              const int& val);
  void remove(int pos);
  int& const  get(int loc) const;
  int& get(int loc);
  void set(int loc, const int& val);
  void push_back(const int& val);
 private:



};
#endif
```

balistint.h

A Bounded Dynamic Array Strategy

- What data members do I need?
  - Pointer to Array
  - Current size
  - Capacity
- Together, think through the implications of each operation when using a static (bounded) array
  - Push_back: Run out of room?
  - Insert: Run out of room, invalid location

```
#ifndef BALISTINT_H
#define BALISTINT_H

class BAListInt {
 public:
  BAListInt(unsigned int cap);

  bool empty() const;
  unsigned int  size() const;
  void insert(int pos,
              const int& val);
  void remove(int pos);
  int const & get(int loc) const;
  int& get(int loc);
  void set(int loc, const int& val);
  void push_back(const int& val);
 private:
  int* data_;
  unsigned int size_;
  unsigned int cap_;

};
#endif
```

balistint.h

Implementation

- Implement the following member functions
  - A picture to help write the code

0	1	2	3	4	5	6	7
30	51	52	53	54	10		

```
BAListInt::BAListInt (unsigned int cap)
{


}

void BAListInt::push_back(const int& val)
{







}

void BAListInt::insert(int loc, const int& val)
{







}
```

balistint.cpp

Implementation (cont.)

- Implement the following member functions
  - A picture to help write the code

0	1	2	3	4	5	6	7
30	51	52	53	54	10		

```
void BAListInt::remove(int loc)
{

}
```

balistint.cpp

Array List Runtime Analysis

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

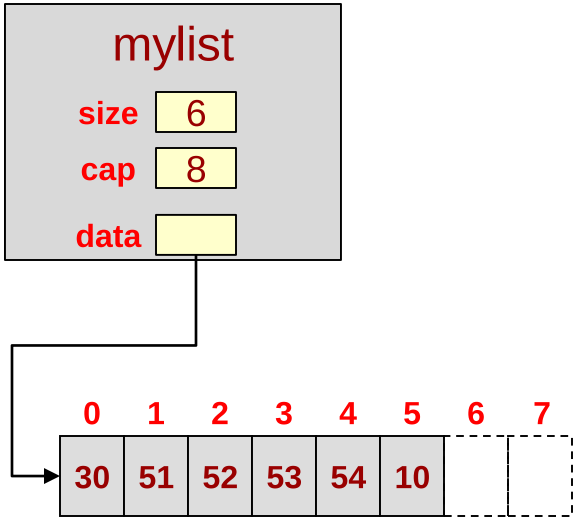
Const-ness

- Notice the get() functions?
- Why do we need two versions of get?
- Because we have two use cases…
  - 1. Just read a value in the array w/o changes
  - 2. Get a value w/ intention of changing it

```
#ifndef BALISTINT_H
#define BALISTINT_H

class BAListInt {
 public:
  BAListInt(unsigned int cap);

  bool empty() const;
  unsigned int  size() const;
  void insert(int pos, const int& val);
  bool remove(int pos);

  int& const  get(int loc) const;
  int& get(int loc);

  void set(int loc, const int& val);
  void push_back(const int& val);
 private:



};
#endif
```

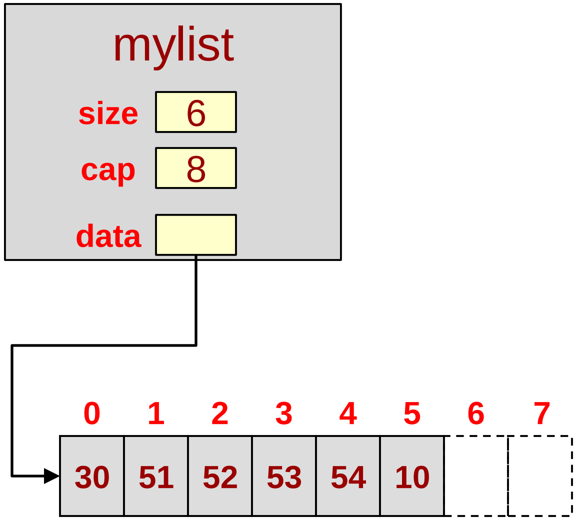
Constness

```
// ---- Recall List Member functions ------
// const version
int& const  BAListInt::get(int loc) const
{ return data_[i]; }

// non-const version
int& BAListInt::get(int loc)
{ return data_[i]; }

void BAListInt::insert(int pos, const int& val);


// ---- Now consider this code ------
void f1(const BAListInt& mylist)
{
  // This calls the const version of get.
  // W/o the const-version this would not compile
  //  since mylist was passed as a const parameter
  cout << mylist.get(0) << endl;
  mylist.insert(0, 57); // won't compile..insert is non-const

}

int main()
{
  BAListInt mylist;
  f1(mylist);

}
```



Returning References

```
// ---- Recall List Member functions ------
// const version
int& const  BAListInt::get(int loc) const
{ return data_[i]; }

// non-const version
int& BAListInt::get(int loc)
{ return data_[i]; }

void BAListInt::insert(int pos, const int& val);

// ---- Now consider this code ------
void f1(BAListInt& mylist)
{
  // This calls the non-const version of get
  // if you only had the const-version this would not compile
  //  since we are trying to modify what the
  //  return value is referencing
  mylist.get(0) += 1;  // equiv. mylist.set(0, mylist.get(0)+1);
  mylist.insert(0, 57);
  // will compile since mylist is non-const
}
int main()
{ BAListInt mylist;
  f1(mylist);
}
```

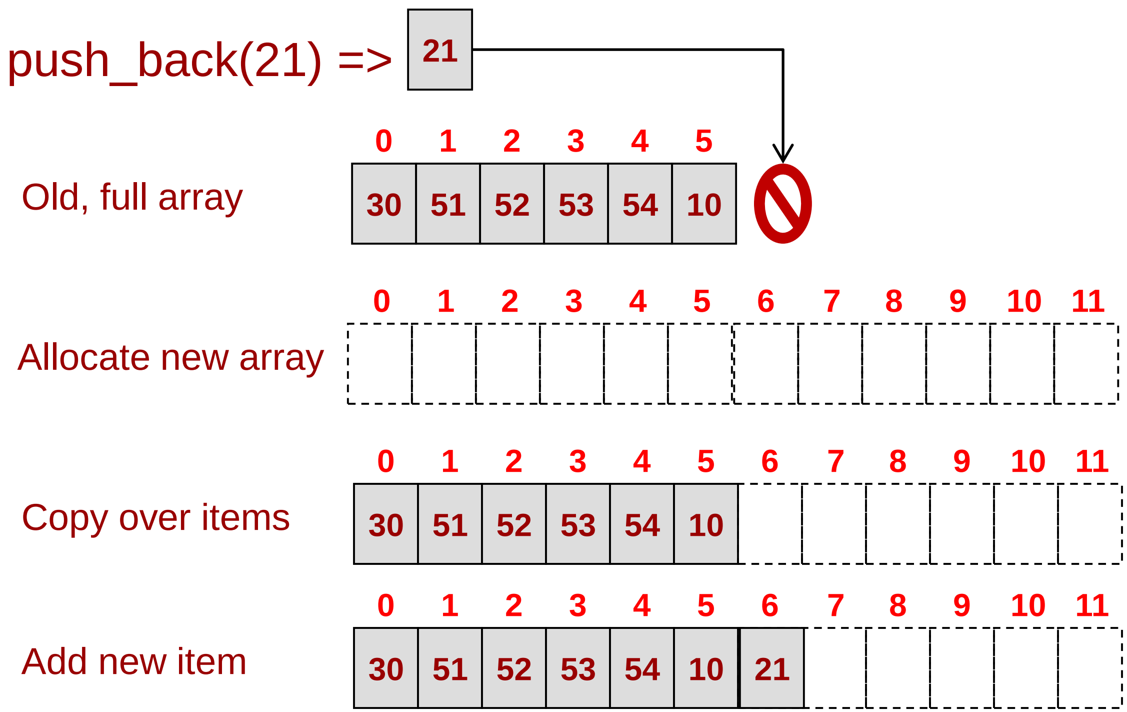


Moral of the Story: We need both versions of get()

UNBOUNDED DYNAMIC ARRAY STRATEGY

Unbounded Array

- Any bounded array solution runs the risk of running out of room when we insert() or push_back()
- We can create an unbounded array solution where we allocate a whole new, larger array when we try to add a new item to a full array



We can use the strategy of allocating a new array ***twice*** the size of the old array

Activity

- What function implementations need to change if any?

```
#ifndef ALISTINT_H
#define ALISTINT_H

class AListInt  {
 public:
  bool empty() const;
  unsigned int  size() const;
  void insert(int loc, const int& val);
  void remove(int loc);
  int& const   get(int loc) const;
  int& get(int loc);
  void set(int loc, const int& val);
  void push_back(const T& new_val);
 private:


  int* _data;
  unsigned int _size;
  unsigned int _capacity;
};

// implementations here
#endif
```

Activity

- What function implementations need to change if any?

```
#ifndef ALISTINT_H
#define ALISTINT_H

class AListInt  {
 public:
  bool empty() const;
  unsigned int  size() const;
  void insert(int loc, const int& val);
  void remove(int loc);
  int& const   get(int loc) const;
  int& get(int loc);
  void set(int loc, const int& val);
  void push_back(const T& new_val);
 private:
  void resize(); // increases array size
  int* _data;
  unsigned int _size;
  unsigned int _capacity;
};

// implementations here
#endif
```

A Unbounded Dynamic Array Strategy

- Implement the push_back method for an unbounded dynamic array

```
#include "alistint.h"

void AListInt::push_back(const int& val)
{











}
```

alistint.cpp

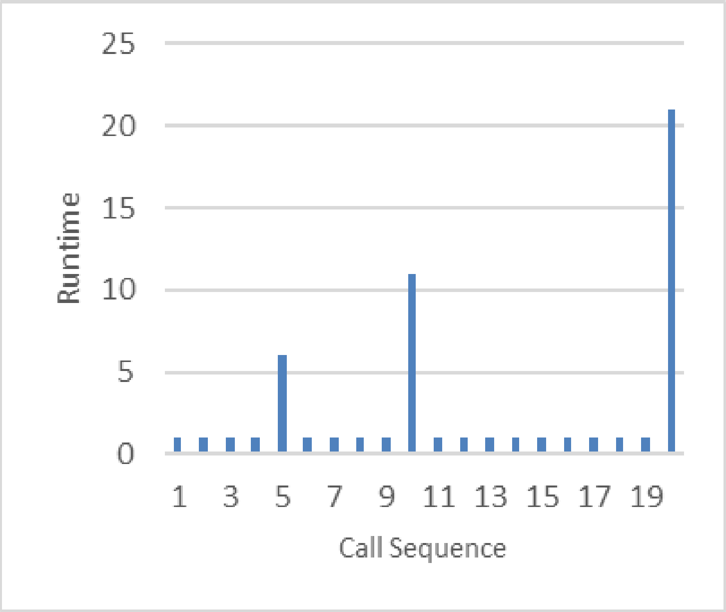
AMORTIZED RUNTIME

Example

- You love going to Disneyland. You purchase an annual pass for \$240. You visit Disneyland once a month for a year. Each time you go you spend \$20 on food, etc.
  - What is the cost of a visit?
- Your annual pass cost is spread or "**amortized**" (or averaged) over the duration of its usefulness
- Often times an operation on a data structure will have similar "irregular" (i.e. if we can prove the worst case can't happen each call) costs that we can then amortize over future calls

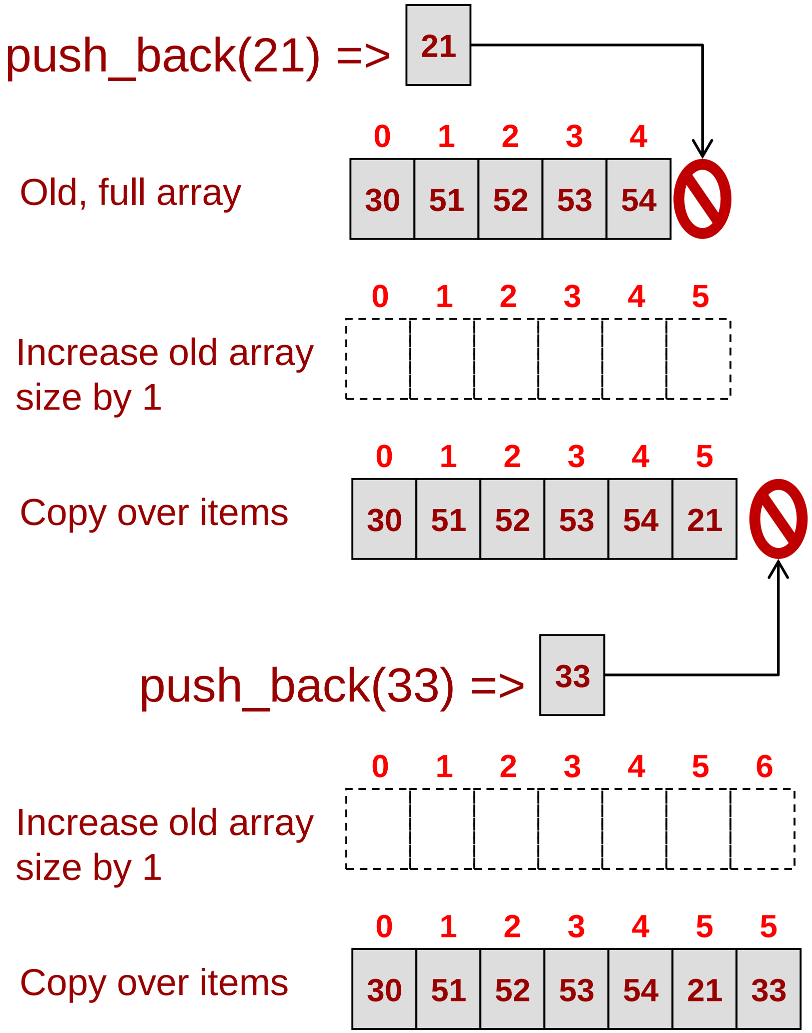
Amortized Run-time

- Used when it is impossible for the worst case of an operation to happen on each call (i.e. we can prove after paying a high cost that we will not have to pay that cost again for some number of future operations)
- Amortized Runtime = (Total runtime over k calls) / k
  - Average runtime over k calls
  - Use a "period" of calls from when the large cost is incurred until the next time the large cost will be incurred



Amortized Array Resize Run-time

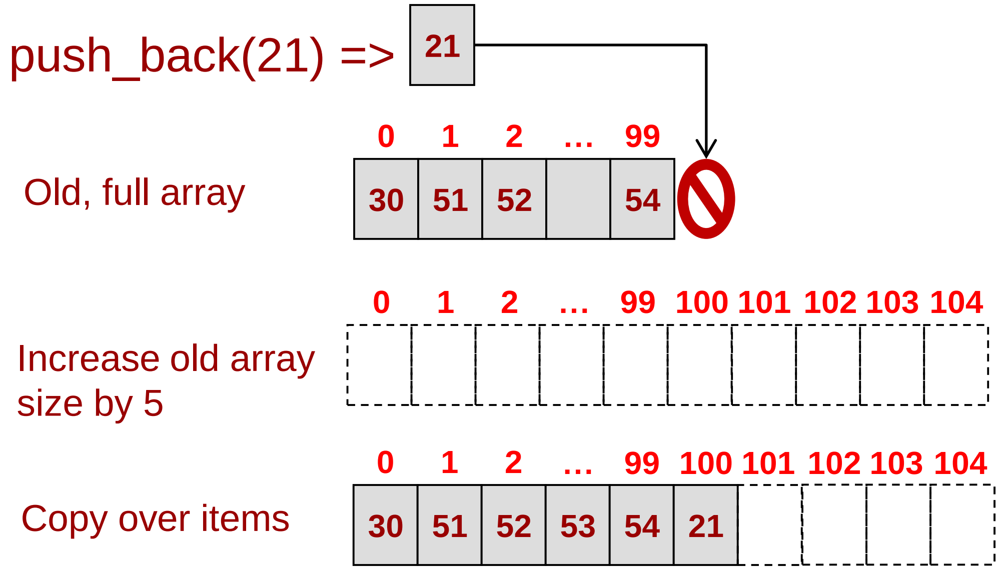
- What is the run-time of insert or push_back:
  - If we have to resize?
  - O(n)
  - If we don't have to resize?
  - O(1)
- Now compute the total cost of a series of insertions using resize by 1 at a time
- Each new insert costs O(n)… not good



Resize by 1 strategy

Amortized Array Resize Run-time

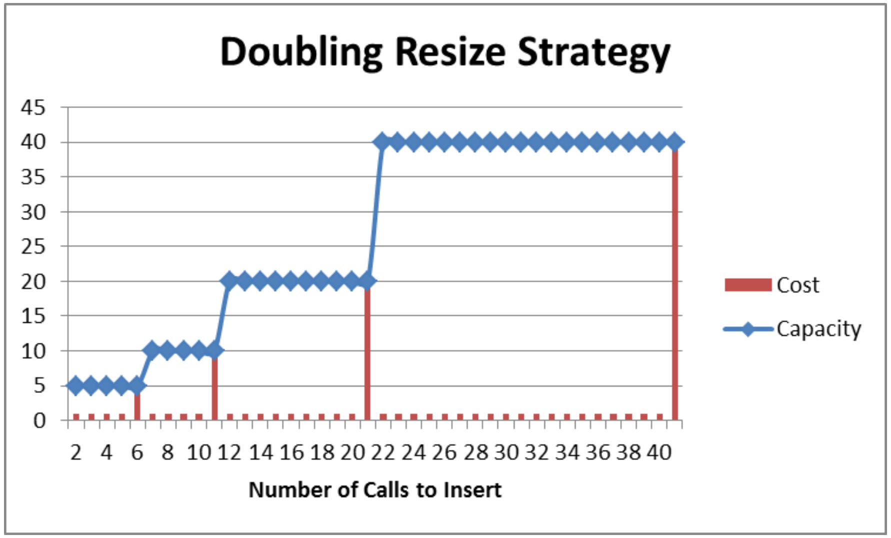
- What if we resize by adding 5 new locations each time
- Start analyzing when the list is full…
  - 1 call to insert will cost: n+1
  - What can I guarantee about the next 4 calls to insert?
    - They will cost 1 each because I have room
  - After those 4 calls the next insert will cost: (n+5)
  - Then 4 more at cost=1
- If the list is size n and full
  - Next insert cost = n+1
  - 4 inserts after than = 1 each = 4 total
  - Thus total cost for 5 inserts = n+5
  - Runtime = cost / inserts = (n+5)/5 = O(n)



Resize by 5 strategy

Consider a Doubling Size Strategy

- Start when the list is full and at size n
- Next insertion will cost?
  - O(n+1)
- How many future insertions will be guaranteed to be cost = 1?
  - n-1 insertions
  - At a cost of 1 each, I get n-1 total cost
- So for the n insertions my total cost was
  - n+1 + n-1 = 2*n
- Amortized runtime is then:
  - Cost / insertions
  - O(2*n / n) = O(2)
    = O(1) = constant!!!



When To Use Amortized Runtime

- When should I use amortized runtime?
  – When it is impossible for the worst case of an operation to happen on each call (i.e. we can prove after paying a high cost that we will not have to pay that cost again for some number of future operations)
  – Generally, a necessary condition for using amortized analysis is some kind of state to be maintained from one call to the next (i.e. in a global variable or more often a data member of an object) that determines when additional work is required
    - E.g. the size_ member in the ArrayList
- Over how many calls should I average the runtime?
  – Determine the period between the worst case occurring (i.e. how many calls between the worst cases occurring)
  – Average the cost over the that number of calls

Example

- What is the worst case runtime of f1()?
  - $T(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} \theta(1) = \theta(n^2)$
- Can the worst case happen each time?
  - No, only every n-th time
- Amortized runtime
  - $\frac{\theta(n^2)+1+\cdots+1}{n} = \theta(n)$

```
int n = // set somehow;
int x = n;

int f1()
{
  if(x == 0){
    for(int i=0; i < n; i++) {
       for(int j=0; j < i; j++){
          // do O(1) task
       }
    }
    x = n;
  }
  else {  x--; }
}
```

Another Example

- Let's say you are writing an algorithm to take a n-bit binary combination (3-bit and 4-bit combinations are to the right) and produce the next binary combination
- Assume all the cost in the algorithm is spent changing a bit (define that as 1 unit of work)
- I could give you any combination, what is the worst case run-time? Best-case?
  - O(n) => 011 to 100
  - O(1) => 000 to 001

3-bit Binary
000
001
010
011
100
101
110
111

4-bit Binary
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Another Example

- Now let's consider an object that stores an n-bit binary number and a member function that increments it (in order) w/ no other way to alter its value
  - Starting at 000 => 001 : cost = 1
  - Starting at 001 => 010 : cost = 2
  - Starting at 010 => 011 : cost = 1
  - Starting at 011 => 100 : cost = 3
  - Starting at 100 => 101 : cost = 1
  - Starting at 101 => 110 : cost = 2
  - Starting at 101 => 111 : cost = 1
  - Starting at 111 => 000 : cost = 3
  - Total = 14 / 8 calls = 1.75
- Repeat for the 4-bit
  - 1 + 2 + 1 + 3 + 1 + 2 + 1 + 4 + …
  - Total = 30 / 16 = 1.875
- As n gets larger…Amortized cost per call = 2

3-bit Binary
000
001
010
011
100
101
110
111

4-bit Binary
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

SOLUTIONS

Doubly-Linked List Prepend

- Assume DLItem constructor:
  - DLItem(int val, DLItem* next, DLItem* prev)
- Add a new item to front of doubly linked list given head and new value

```
void prepend(DLItem *& head,
             int n)
{
  DLItem* elem = new DLItem(n, head, NULL);
  head = elem;
  if (head->next != NULL){
    head->next->prev = head;
  }
};
```

Doubly-Linked List Remove

- Remove item given its pointer

```
void remove(DLItem *& head, DLItem *splice)
{
  if (splice != head){
    splice->prev->next = splice->next;
  }
  else {
    head = splice->next;
  }
  if (splice->next != NULL){
    splice->next->prev = splice->prev;
  }
  delete splice;
}
```

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	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	1 pointer (next)
Singly linked-list w/ head and tail ptr	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	1 pointer (next)
Doubly linked-list w/ head and tail ptr	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	2 pointers (prev + next)

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