Lists

Betsy covered: The list ADT, isempty/erase/insert/find/remove/moveToEnd, algorithms, how to improve slist by adding tail/cursor/dummy node/doubly LL,

IDEA FOR NEXT TIME:

Give handout with space for them to fill in C++ or pseudocode for each function we write.

the handout will have blanks.

the handout is a C++ class with space for each function. (kind of like a worksheet) we fill it out together as class goes on.

**Review vector ADT**

* Description: A vector consists of a collection of positions, each of which contains a single element of the vector. Each position has a unique index, which is an integer in the range 0 <= index < n, where n = # of elements in the vector.
* Note that this says nothing about how the vector is actually stored in memory. All it says is there’s a correspondence between integers and the elements of the array. We are free to pick whatever implementation makes most sense in this context.
* Review Big-oh times of Vector operations.
* In particular, focus on insertion being an O(n) operation.
* The List ADT focuses in improving insertion/deletion operations, at the expense of losing the efficiency of random access.

**The List ADT**

* Description: A list consists of a collection of positions in linear order, each of which contains a single element of the list.
* Note: the only thing really different between the Vector ADT and the list ADT is that for lists, we don’t emphasize the position indices in the description, which implies that this ADT doesn’t focus on random access.

**Ask what operations a LIST should be able to do?**

* Insert an item at the beginning/end/middle.
* Remove an item in the beginning/end/middle.
* Retrieve the first/last item.
* Search for an item.
* Iterate through the list.
* Get the size of the list.

Big idea of lists vs vectors: In choosing between these two ADTs, you should ask yourself, do you prioritize fast random access or fast insertion/deletions? If you want fast random access, choose a VECTOR implementation. If not, choose a LIST, usually implemented with some sort of linked list structure.

**Review the highlights common to all linked lists.**

* Recall from 142 to the basic idea of a linked list.
* We have a structure called a NODE. Nodes are linked together through pointers. **DRAW PIC.**
* Explain, using pictures, why insertion and deletion is easy (no sliding elements back and forth in an array).
* But notice that we can’t skip to any arbitrary element in an easy manner, thus no FAST random access.
* Preview of what’s to come: those two operations in lists vs vectors swap big oh efficiencies. Insertion/deletion in a linked list become constant time (whereas they were linear time in vector), but accessing an arbitrary element becomes linear time (whereas it was constant before).

**Explain some of the different types of linked lists.**

* Different types of linked lists can be distinguished by (1) how many pointers each node structure has to other nodes, (2) how the nodes are linked together (shape), and (3) how many “special” pointers we keep around that always point to certain nodes within the list.
* (1) The simplest kind of linked list uses a Node structure with one pointer, which we think of as pointing to the “next” item in the list.
  + This is called a singly-linked list.
* We also have a Node structure with two pointers, that point to the previous and next items.
  + Called a doubly-linked list.
  + As we mentioned earlier in class, a 2x linked list lets you traverse the list in either direction quickly. The 1x linked list only allows you to go forwards.
* There are other also other kinds of lists that use more than 2 pointers, such as a skip list.
* In regards to (2), normally we think of a list as having a beginning and an end, but sometimes we will make circularly-linked lists.
* And (3), when we actually implement linked lists, we need a certain number of pointers that will always give us access to certain parts of the list.
  + At a minimum, we need a pointer to the first item in the list. Even if we only have a singly-LL, a pointer to the first item in the list will let us get to all the other items.
  + Lots of times, we will also keep around a pointer that will always point to the TAIL of the list.
    - **Ask**: what does this accomplish? [a tail pointer in a SLL]
    - Answer: allows us to insert things at the end of the list in O(1) time, rather than O(n) time.

So today we will explore the simplest type of LL, a singly-linked list where we only have a pointer to the first item in the list.

**Write the node structure.**

* Linked lists, like arraylists, are CONTAINERS, in that they hold other data types.
* So when you use a linked list, you must know ahead of time what type of data it’s going to store, be it ints, chars, strings, or some other object.
* So simplify things, we’ll assume that our linked lists today will hold integers.
* Normally, to implement a SLL, a node holds only two components, an item of the list (in our case, an int), and a pointer to the next node. We implement these in Java with a class.
* class Node
* {
* public int data; // sometimes called value or item   
   public Node next = null; // explain what this does  
  }
* Explain why these are public (at least for me): Lots of times the code to manipulate these Node objects exists outside the Node class itself.
* For instance, often times there is a special pointer to the first Node in a linked list. This node is often called head (or first).
* Draw this as a linked list. (Head pointer which points to the starting node, 1—2—3)

**LL Algs – emphasize drawing pictures.**

**Talk about traversals**

* Traversal means going through the linked list in a sequential manner. Depending on the list, this can be front to back or back to front.
* No matter how the list is implemented, (1x, 2x, circular), it always works in pretty much the same manner.
* To traverse a list in the forward direction:
* curr = (wherever you want to start traversing in the list), e.g., HEAD.
* while (stopping condition not reached) // sometimes people will put while(curr)
* {
* // process curr.data
* // advance the curr pointer 🡺 curr = curr.next
* }

Compare/contrast with looping over an arraylist with a for loop.

* What are stopping conditions?
  + End of list: use curr != null
  + Looking for a specific item: use curr.data != WHATEVER.

**Choosing implementation details for a Java class**

* What items should we have in our class?
* Certainly a pointer to the first item in the list.
* (Pointer to last element if we want it)
* (Pointer to current or active element if we want it)
* (INT SIZE) --- what does this do?
  + size() function – if we have this int, this is O(1), otherwise O(n).

**Insert**

**Insert After a node:**

* Assume we have traversed a list so that we have a pointer, called curr. And we want to insert a new node after curr.
* Draw picture 10, 20, 30, 40, 50.
* Assume we have a pointer to 30.
* Draw picture of adding 35 after 30.
* Allocate a new node:
* Allocate new node to insert.
  + newnode = new node;
  + newnode.data = [whatever]
* Link up the nodes
  + newnode.next = curr.next // this must be first, line below must be 2nd
  + curr.next = newnode

**Insert Before a Node**

* What if we want to insert something BEFORE the 30?
* Draw picture of insertion.
* Note that we NEED a pointer to the previous node (before 30) but we don't have it.  
    
  The problem is during traversal of a linked list, once we find the node we're looking for, we can't go back.
* So we use a 2nd type of traversal, where we maintain TWO pointers. One pointer is still called curr, the other trails behind curr, and we called it prev (for previous).
* To traverse a list in the forward direction:
* curr = head (wherever you want to start traversing in the list)
* prev = null (previous node from curr, if you have it, or null if beginning of list)
* while (stopping condition not reached)
* {
* // process curr.data  
   // advance the prev pointer =🡺 prev = curr [// prev=prev.next doesn’t work for null]
* // advance the curr pointer 🡺 curr = curr.next
* }
* So back to **inserting BEFORE**.
* Allocate new node to insert.
  + newnode = new node;
  + newnode.data = [whatever]
* Link up the pieces (2 pointers to change)
  + newnode.next = curr;   
    prev.next = newnode
* What if prev doesn’t exist? inserting at front!  
  + Allocate newnode as before.
  + newnode->next = head
  + head = newnode

**Delete**

* Assume we have traversed the list so that we are deleting curr
* prev->next = curr->next
* OR IF WE are deleting first item in list:
* head = head->next

These are the two basic cases for deletion in a SLL.

This correctly handles deleting a list with one element.

**Clear/Erase**

* Just set head = null

**Copy** //phrased as copying into ourselves from an outside list (otherlist)

Have students do this as a class:

Suggestions:

handle an empty list

if list is not empty, copy the head node first,

then copy the rest of the list (maintain two pointers that walk down the old list and the new list).

BASIC ITERATION:

curr = head

while (curr != null) {

// process

curr = curr.next

}

NEW PART:

*// Copy head manually.*Node head\_newlist = new Node();  
head\_newlist.data = head.data;  
Node prev\_newlist = head\_newlist;  
  
Node curr = head.next; // already copied head, so skip it  
while (curr != null)  
{  
 Node curr\_newlist = new Node();  
 curr\_newlist.data = curr.data;  
 prev\_newlist.next = curr\_newlist;  
 prev\_newlist = curr\_newlist; *// advance newlist* curr = curr.next; *// advance this list*}

DAY 2 REVIEW

* Mention what we did last time:
  + 1 pointer traverse, insert after, 2 pointer traverse.
* Review code for all of these.  
    
  1 pointer traverse:

curr = head

while (curr != null) { // or whatever condition you want

// process

curr = curr.next

}

2 pointer traverse

curr = head (wherever you want to start traversing in the list)

prev = null (previous node from curr, if you have it, or null if beginning of list)

while (curr != null) // or whatever condition you want

{

// process curr.data  
 // advance the prev pointer =🡺 prev = curr [// prev=prev.next doesn’t work for null]

// advance the curr pointer 🡺 curr = curr.next

}

* Insert after:
  + Assume we have a curr pointer, we want to insert something after it.
  + Allocate new node to insert.
    - newnode = new node;
    - newnode.data = [whatever]
  + Link up the nodes
    - newnode.next = curr.next // this must be first, line below must be 2nd
    - curr.next = newnode
* So where we left off before was we had a list of 10, 20, 30, 40, 50, we had a pointer to 30, and we want to insert 25 before it.
* So back to **inserting BEFORE**.
* Allocate new node to insert.
  + newnode = new node;
  + newnode.data = [whatever]
* Link up the pieces (2 pointers to change)
  + newnode.next = curr;   
    prev.next = newnode
* What if prev doesn’t exist? inserting at front!  
  + Allocate newnode as before.
  + newnode.next = head
  + head = newnode
* **DELETION**
* Assume we have traversed the list so that we are deleting curr
* prev->next = curr->next
* OR IF WE are deleting first item in list:
* head = head->next

These are the two basic cases for deletion in a SLL.

This correctly handles deleting a list with one element.

Cover big-oh of these operations.

* All of the insertions – none use loops. O(1).
* All of the deletions – O(1).
  + Caveat – time to traverse the list to find the right place to insert: O(n).
* Make this part explicit:
* SLL with only head pointer: [draw picture]
  + insert at head, O(1).
  + insert at tail, O(n). [takes time to traverse]
  + delete at head, O(1).
  + delete at tail, O(n) [must traverse]
  + IN GENERAL: insert/delete anywhere is O(1) + traversal time.
* SLL with head and tail pointers: draw picture.
  + insert at head, O(1).
  + insert at tail, O(1).
  + delete at head O(1).
  + delete at tail, O(n). Need to traverse to get prev node.
  + IN GENERAL insert/delete anywhere, O(1) + traversal time.
* Do this for array:
  + insert at head – must shift over, O(n)
  + insert at tail – worst case must reallocate O(n)
  + delete at head – must shift over, O(n)
  + delete at tail – no shift O(1)
  + ACCESS any position O(1)
* For both types of SLL, access anything where you must traverse is O(n).
* So here's the takeaway:
  + If your application requires more insertions and deletions than random accesses, use a linked list.
  + If your application requires more random accesses than insertions and deletions, use an arraylist.
  + By default, steer towards arraylist. These are heavily optimized in most languages, and most of the time, you end up inserting things at the end.

DAY 2 – new material

* Major topics:
* singly – finish singly LL - need to cover delete
* doubly linked lists (introduce, operations: insert, delete)
* stacks & queues?
* Introduce new Node class
  + int data
  + Node next
  + Node prev
* insert (after a node)
  + assume we want to insert after a pointer called curr
  + make new node and link up
  + node newnode = new node
  + newnode.next = curr.next
  + newnode.prev = curr
  + curr.next.prev = newnode
  + curr.next = newnode
* insert (before a node)
  + assume we want to insert BEFORE a pointer called curr
  + make new node and link up
  + node newnode = new node
  + newnode.next = curr
  + newnode.prev = curr.prev
  + curr.prev.next = newnode
  + curr.prev = newnode
* Caveats – insertion is more complicated when inserting at the head and tail, because you may have to modify the head pointer, and tail pointer if you have one (most often doubly linked lists have both a head and tail pointers).
* Deletion – Easier than insertion, but requires special handling of head and tail pointers.
* Basic operations for delete:
  + Suppose we want to delete the node pointed to by curr.
  + curr.prev.next = curr.next
  + curr.next.prev = curr.prev
* But we also need to check BEFORE any of this if curr.prev is null (in that case we are the 1st elt) and if curr.next is null, in which case we are the last element.
  + if curr.prev == null // curr is head
    - head = curr.next
  + else
    - curr.prev.next = curr.next
  + if curr.next = null // curr is tail
    - tail = curr.prev
  + else
    - curr.next.prev = curr.prev
* Big oh times
* Operation Singly(head) Singly(head/tail) Doubly
* Insert-head O(1) O(1) O(1)
* insert-tail O(n) O(1) O(1)
* insert-middle O(1) + traverse O(1) + traverse O(1) +traverse
* delete-head all O(1)
* delete-tail O(n) O(n) – ask stu why? O(1)
* delete-middle O(1) + trav O(1) + trav O(1) + trav
* search (for data) O(n) O(n) O(n)
* random access(posn) O(n) O(n) O(n)

For arraylist (RArrayList):

insert-

IN JAVA – Use LinkedList and ArrayList.

**Day 3**

* Review stacks
* queues
* Deques
* Recursive algs for LLs
* Special delete algorithm?

**stacks:**  
A Stack is an ADT that is a restricted version of a list. [Remember ADT=abstract data type, tells us what this type is capable of doing but not how it does it.] We know that for a List abstract data type, we normally have the ability to add elements anywhere we want within a list, and remove elements wherever we want, at any time.

Stacks are different. A Stack is capable of adding and removing items, but only from one end of the list.

Analogy as to why it's called a Stack. Imagine you're in a cafeteria like the rat, and you go to any of the serving lines, there's a stack of plates there. In a stack of plates, you can (easily) only access the top plate. You can either put another plate on top of the stack, or you can take the top plate off. There's no easy to way to insert or remove plates anywhere else in the stack.

aka LIFO = last in, first out.

TWO main operations:

* Push – add an element to the top of the stack.
* Pop – Remove & return the top element on the stack.
* [That's everything that a stack \*must\* do.]
* Other things usually included:
* Peek – Return (but don’t remove) the top element on the stack. [sometimes called top]
* Usually also includes isEmpty() [check if stack is empty]
* *All operations that modify a stack happen at the SAME END.*

Let's review these operations in the abstract. So imagine a stack of plates in your head. Often drawn vertically.

Review operations: (draw these vertically as they happen)

* PUSH 1,2,3,4,5, POP 5 times => 5 4 3 2 1
  + Things naturally come out in reverse order that you push them.
* Example (mixing pushes and pops):
  + PUSH 1
  + PUSH 2
  + PUSH 3
  + POP
  + PUSH 4
  + POP
  + POP
  + PUSH 5
  + POP
  + POP
  + 🡺 3 4 2 5 1
* Sometimes called a LIFO stack. (last in, first out)

Now for the actual implementation, how would you build this? Can implement with an array or LL:

* With array, usually push and pop happen at the “right end” of the array because no shifting is necessary.
* With LL, usually push/pop happen at the “left end” (head) because then you only need a head pointer, no tail pointer.
* OPERATION IN ARRAY/VECTOR IN LL
* Push insert at end (rhs) insert at head
* Pop remove at end remove at head
* For a LL, do we need a singly LL or a doubly linked list? Remember, all we need to do is push/pop. [SLL is fine]
* If implementing from scratch with an array, often what people do, especially if they know there will be a maximum # of elements on the stack, they allocate the largest possible stack all at once. Avoids expanding/copying.

To be clear, using a stack in an algorithm is just like using a an Arraylist/LL where you make sure to only add and remove things from one end. But we give it its own name and ADT because it’s so incredibly common in algorithms.

We also give it its own name because as a restricted form of a full-fledged list, it is also a signal to us as programmers as to the important operations. If we are implementing an algorithm, and it calls for a LIST, we will probably need to read through the algorithm and determine out how that list is being used before we can choose whether this LIST should be an arraylist or a SLL, or a DLL, or maybe even something else. Whereas if the algorithm specifies this is a STACK, we know aha, ok, all the insertions and removals only happen at one end, I probably don't need a DLL b/c that's overkill.

Part of the reason why stacks get their name is also historical. They were first conceived of in the early days of computers and programming languages, because they show up in memory allocation algorithms and interpreters and compilers for programming languages. So they got their own separate name because they were so important in the early days of computing.

As an example, how many of you know about stack overflow? How many of you, when writing a recursive function, have ever gotten a stack overflow error?

So that name comes from the fact that the way most programming languages implement function calls is with a stack.

* Every time a function is called, this thing called an ACTIVATION RECORD is pushed onto a special stack inside the programming language.
* An activation record is in charge of keeping track of all of a function's local variables, among other things.
* So every time a function is called, a new activation record is created and pushed on to the stack.
* Show example for main, which calls f(), which calls g().
* Every time a function ends, it pops the current activation record off the top, and the one below becomes the top of a stack.
* If you've ever wondered how a program knows when a function ends, how to jump back to the function where it was called from, this is exactly how. All that info is stored in these records, and so when a function ends and its activation record is popped off the stack, the next one comes up to the top and the program just picks back up with that one.
* Do iterative example 🡪 main() calls f(), calls g().
* Do recursive example -> main() calls fact().
* So where are these used in CS, besides in implementing programming languages?
* **Web browser history**. Every time you go to a new page, it's pushed onto the stack, and when you hit back, it's popped off.
  + Now in truth, nowadays its probably a list, because you can see the whole browser history, but in the early days, there was just a back button, and you couldn't hold it down to see the whole history.
* UNDO operation in applications. When you do an operation, instructions to undo it are pushed onto a stack. And when you click the undo button, they are popped off and executed.
* They also show up in other places in programming languages besides function calls. For example, when determining if a program is legal code, you have to look for balanced parentheses, balanced braces, etc. Each opening one has to have a matching closed one. The usual way this is implemented is with stacks, where, for instance, every time you see an opening parentheses, you push it onto a stack, and every time you see a closing one, you pop it off. The only way the program can have balanced parentheses is if you can parse the whole program this way, and then you're guaranteed every opening one has a closing one and vice versa.
* Evaluate RPN. (postfix notation)
  + Alg: Init empty stack.
  + Repeat until end of expr
    - Get next token.
    - If is number, push onto stack.
    - If operator, pop stack twice to get numbers.
    - Evaluate
    - Push result onto stack.
  + Evaluate: 2 4 \* 9 5 + - => -6

QUEUES

Intro: Queues are complementary to stacks.

* Whereas stacks add and remove items from the same end, queues add to one end and remove from the other.
* So the visual metaphor for a queue is a WAITING LINE. People always join the line at the back, and are served from the front.
* So they naturally show up in algorithms that are maintain an ordered list of things to be processed --- often times when the number of things to be processed is growing at a different rate than the rate at which we process them.

Whereas stacks we normally visualize vertically, we usually visualize queues horizontally.

Queues have a FRONT (left) AND BACK (right).

Sometimes, like lists, these are called HEAD and TAIL.

TWO main operations:

* Enqueue – add an element to the BACK of the queue.
* Dequeue – Remove & return the FRONT element in the queue.  
    
  Plus
* Peek – Return (but don’t remove) the front element in the queue. (also called front)
* Usually also includes isEmpty() [check if queue is empty]
* *Adding is ALWAYS at back. Removing is ALWAYS at front.*

Review operations: (draw horizontally)

* ENQUEUE 1,2,3,4,5, DEQUEUE 5 times => 1 2 3 4 5
  + Things naturally come out in SAME order that you enqueue them.
* Example (mixing pushes and pops):
  + ENQ 1
  + ENQ 2
  + ENQ 3
  + DEQ
  + ENQ 4
  + DEQ
  + DEQ
  + ENQ 5
  + DEQ
  + DEQ
  + 🡺 1 2 3 4 5

So sometimes this is called a FIFO structure.

Again, can be implemented either with arrays or with LLs.

Array-based queues:

* What is the major problem with implementing a queue with an array that we didn’t have with stacks?
  + Ans: we have to shift things, whenever we DEQUEUE from front.
* Possible solution:
  + Maintain two indices (front and back) which store indices into the array of the current front element and THE NEXT back element. Don’t shift!
  + Show example (array of size 5):
    - f = 0, b = 0. ENQ 70
    - f = 0, b = 1. ENQ 80
    - f = 0, b = 2. ENQ 50
    - DEQUEUE twice (removes 70 & 80)
    - ENQUQUE 90 & 60.
  + Before we can add more, we either must expand the array, or shift all the elements back to the beginning.
* If we can guarantee that the QUEUE will never hold more than a MAX number of elements, we can use a “circular array.”
  + Show ENQUEUEING three things, DEQ 2, then ENQUEUE 3.
* How to increment back?
  + back++ for normal
  + back = (back + 1) % CAPACITY
* How to check when queue is empty for non-circular array? front==back.
  + If circular? Same thing?
  + But now we can’t distinguish between an empty queue and a full queue.

Queues as linked lists

* Ask students, what type of list makes sense here, SLL or DLL? Where should the head be, and where should the tail be?
  + Ans: SLL is ok.
* Use head and tail.
* FRONT = head. BACK = tail.
* Dequeue at FRONT (head).
* Enqueue at BACK (tail).
* Why is singly-linked list ok here?
  + B/c we never remove from BACK/TAIL.
* Circular linked list:
  + pointer to head/pointer to tail.
  + Pointer to only tail – can get head by following tail->next.
* Advantage here is we save space – we only need one pointer instead of two.
* Seen in scheduling applications where you constantly rotate through resources so nobody gets overscheduled.
  + Computers for processing jobs.
  + Circular list of computers. Maintain a current pointer. Assign job to current computer. Then move current pointer to next. Take out if full. Add in when free.
* Applications in OS: buffers (keyboard buffer, disk buffer --- reading from hard drive) Network buffer (data arrives over the wire but you may not be ready to process it yet).
* Touch on priority queue if desired./DEQUE

Recursive processing of LL.

void proc(node \* curr)

print curr->data

proc(curr->next)

VS

void proc(node \* curr)

proc(curr->next)

print curr->data