5.2 linked lists

Thursday, September 29, 2022

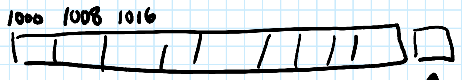
Reminder: pick a partner for lab 5.

Today: linked lists

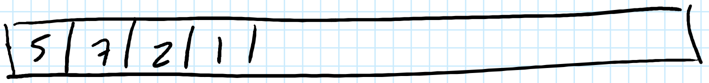
OPERATION	ARRAYLIST runtime
int getSize()	O(1)
bool isEmpty()	O(1)
T get(i)	O(1)
T getFirst()	O(1)
T getLast()	O(1)
void insertFirst(T)	O(n)
void insertLast(T)	{ O(n) worst { O(1) amortized
T removeFirst()	O(n)
T removeLast()	O(1)

Why is Arraylist necessarily slow ?

An array is a block of contiguous memory.

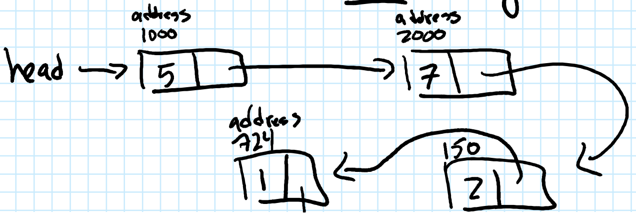


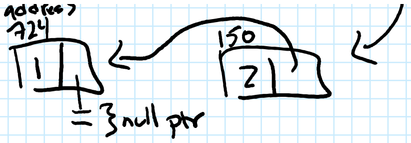
to add one more el, it needs to go in this memory address, which might be already allocated to something else, so we just have to allocate all new memory to get a big-enough contiguous block



Contiguous memory: pro : indexing is O(1)
con: resizing requires a new allocation & copying everything over

Alternative idea: non contiguous memory



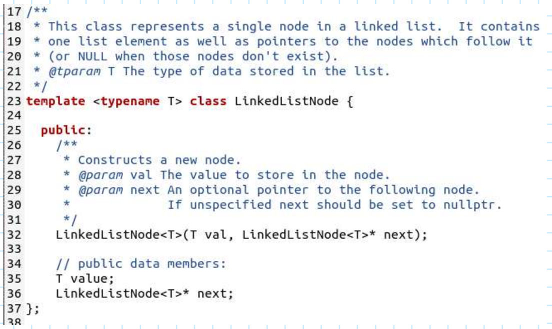


Linked List

A linked list is a series of nodes; each node contains a value and a pointer to the next node in the list.

LinkedListNode declaration

see file "linkedList.h" in your lab 5 repo



```
/**
 * This class represents a single node in a linked list.  It contains
 * one list element as well as pointers to the nodes which follow it
 * (or NULL when those nodes don't exist).
 * @tparam T The type of data stored in the list.
 */
template <typename T> class LinkedListNode {

  public:
    /**
     * Constructs a new node.
     * @param val The value to store in the node.
     * @param next An optional pointer to the following node.
     *             If unspecified next should be set to nullptr.
     */
    LinkedListNode<T>(T val, LinkedListNode<T>* next);

    // public data members:
    T value;
    LinkedListNode<T>* next;
};
```

All data is public since LinkedListNode is not a standalone class, it just gets used inside LinkedList.

LinkedList declaration

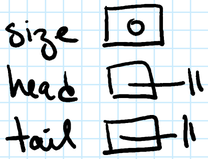
private data :

int size

LinkedListNode<T>* head ← pointing to the node containing the first el

LinkedList Node<T>* tail ← pointing to node containing last el.

Linked List constructor



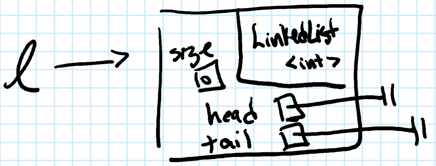
} both null pointers

Linked List insertFirst pseudocode:

```
void insertFirst(T value):
    create a new node to hold
        value, whose next points
        where head points
    update head to point to new node
```

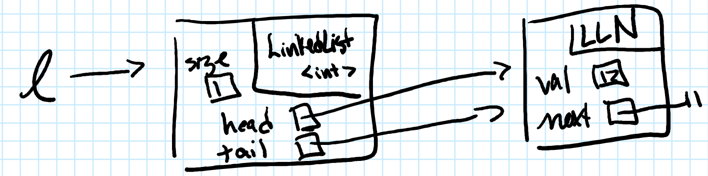
example:

List* l = new LinkedList<int>();

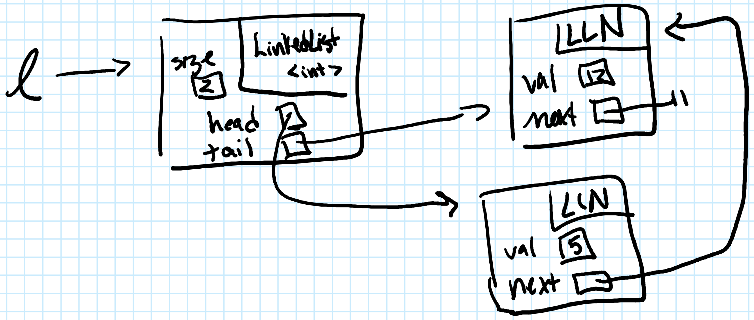


if size = 0, point tail to new node
size ++

l→ insertFirst(12);



l → insertFirst(5);



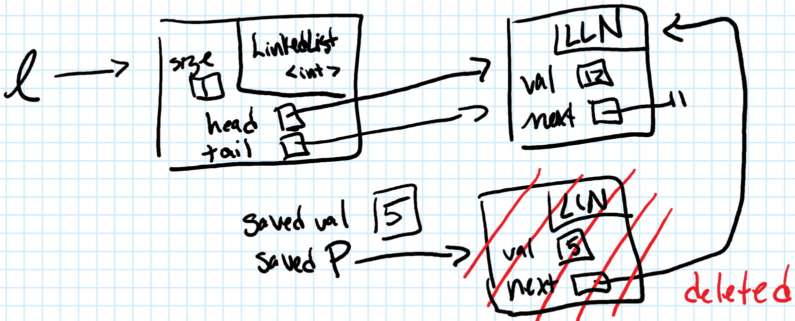
runtime: O(1)

does not depend on size of list

LinkedList removeFirst pseudocode

```
T removeFirst():
    if size = 0, throw runtime error
    save value at head node
    if size = 1:
        | delete the node
        | set head and tail to null pointer
    else:
        | save pointer to head
        | move head forward to next node
        | delete saved pointer
    size --
    return saved value
```

this→head = this→head→next;



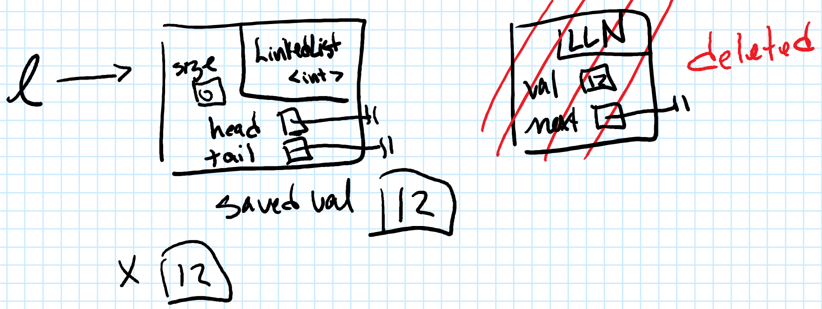
runtime: O(1)

int x = l→ removeFirst();

x $\boxed{5}$

x 15

x = l -> removeFirst();



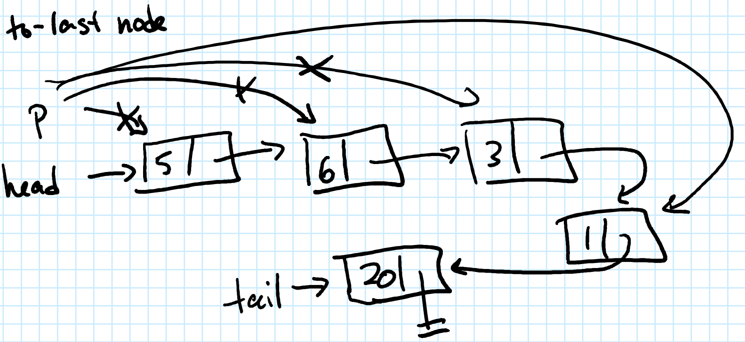
LinkedList removeLast pseudocode

idea: we need to return the value at tail and remove the final node

cases: (A) size = 0 : throw runtime error

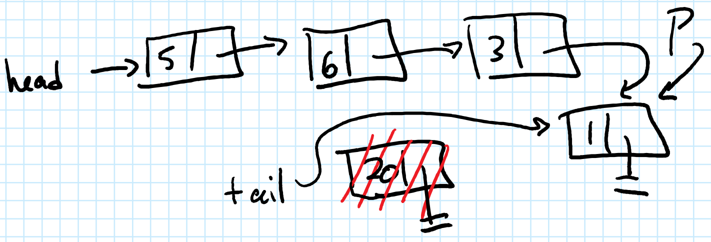
(B) size = 1 : delete the node
head = null ptr
tail = nullptr

(C) size ≥ 2 : need to walk down the list to find the second-to-last node



case C pseudocode:

```
save tail value
p = head
// walk down the list to
// find the second-last node
while ( p -> next -> next != null ptr ):
    p = p -> next   // advance p down the list
// p is now pointing to second-last node
p's next = nullptr
delete tail node
tail = p
decrement size
return saved value
```



runtime: O(n) because we have to

runtime: O(n) because we have to walk down the entire list

Note: if we made a doubly linked list where each node has pointers to previous & next then removeLast would be O(1).

LinkedList checkInvariants

Every data structure we make will have a checkInvariants() method to make sure the data structure is working as expected.

For ArrayList, checkInvariants made sure that $\begin{cases} size \geq 0 \\ size \leq capacity \end{cases}$.

What should checkInvariants for LinkedLists do?

If size = 0, head & tail are null

If size > 0, walk down the list while counting to verify that size is the # of nodes in the list.

} O(n) runtime b/c of walking the entire list.

OPERATION	ArrayList runtime	LinkedList runtime
int getSize()	O(1)	O(1)
bool isEmpty()	O(1)	O(1)
T get(i)	O(1)	O(n)
T getFirst( )	O(1)	O(1)
T get Last( )	O(1)	O(1)
void insertFirst(T)	O(n)	O(1)

T getLast()	O(1)	O(1)
void insertFirst(T)	O(n)	O(1)
void insertLast(T)	O(n) worst O(1) amortized	O(1)
T removeFirst()	O(n)	O(1)
T removeLast()	O(1)	O(n)

So... which subclass is better?

Depends on your application. Lots of insert/remove from front : LL better

lots of indexing : ArrayList better.

Why bother with ArrayLists when we can directly use arrays?

Easier for user : details hidden, size checking done for you, errors useful.

Templated!

C++ standard template library

A library of generic container classes.
(templated!)

simple example: a pair

(convenient way of joining two things of any type together into a unit)

```
pair < int, string> student;
student.first = 2019;
student.second = "Douglas";
pair < string, float> item;
item.first = "widget";
item.second = 7.6;

cout<< student.first <<endl; //prints "2019"
cout << item.second << endl; //prints "7.6"
```