Review of Data Structures

Virendra Singh
Professor, Indian Institute of Technology Bombay
And
Adjunct Professor, Indian Institute of Technology Jammu
http://www.ee.iitb.ac.in/~viren/
E-mail: viren@ee.iitb.ac.in, virendra.singh@iitjammu.ac.in

CSPL201: Data Organization & Retrieval

Lecture 4 A (23 September 2020)

भारतीय प्रौद्योगिकी संस्थान जम्मू
INDIAN INSTITUTE OF TECHNOLOGY JAMMU

Acknowledgement

- Prof. Sartaj Sahni, Uni. of Florida
- Prof. Hideo Fujiwara, NAIST
- Late Prof. A. Bhattacharya, SERC, IISc

Data Structures

- Data structure is a way to store and organize data in order to facilitate access and modification
- No single data structure works well for all purposes

- data object
- set or collection of instances
  - integer = {0, +1, -1, +2, -2, +3, -3, …}
  - daysOfWeek = {S,M,T,W,Th,F,Sa}
- instances may or may not be related
  - myDataObject = {apple, chair, 2, 5.2, red, green, Jack}

Data Structure

Data object

relationships that exist among instances

and elements that comprise an instance

- Among instances of integer

  369 < 370

  280 + 4 = 284

- The relationships are usually specified by specifying operations on one or more instances.
  - add, subtract, predecessor, multiply

Linear (or Ordered) Lists

- instances are of the form
  - $(e_0, e_1, e_2, \ldots, e_{n-1})$
  - where e_i denotes a list element
  - $n >= 0$ is finite
  - list size is n
- $L = (e_0, e_1, e_2, e_3, \ldots, e_{n-1})$
- relationships
  - e_0 is the zero’th (or front) element
  - e_{n-1} is the last element
  - e_i immediately precedes e_{i+1}

Linear List Examples/Instances

Students in EE717 =

(Deepak, Jaidev, , Amit, Abhishek, …, Vijay)

Exams in EE717 =

(Test1, Midsem, Test 2, Test 3, Final)

Days of Week = (S, M, T, W, Th, F, Sa)

Months = (Jan, Feb, Mar, Apr, …, Nov, Dec)

Linear List Operations

- Size ( )
  - determine list size
  - *L = (a,b,c,d,e)*
  - size = 5
- Get (theIndex)
  - get element with given index
  - *L = (a,b,c,d,e)*
  - *get(0)* = *a*
  - *get(2)* = *c*
  - *get(-1)* = error
  - *get(9)* = error

Linear List Operations

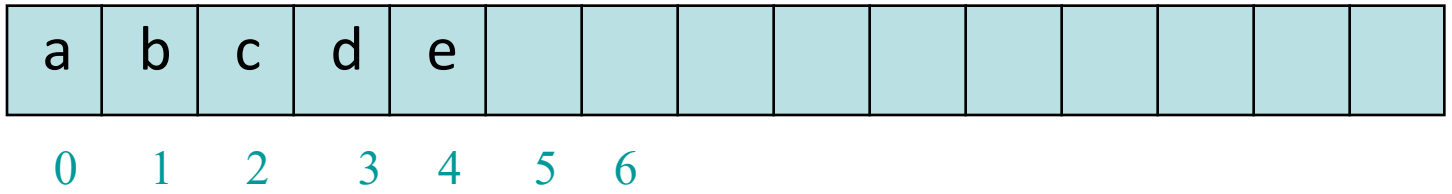
- IndexOf(theElement)
  - determine the index of an element
  - *L = (a,b,d,b,a)*
  - *indexOf(d) = 2*
  - *indexOf(a) = 0*
  - *indexOf(z) = -1*
- Remove(theIndex)
  - remove and return element with given index
  - *L = (a,b,c,d,e,f,g)*
  - *remove(2)* returns *c*
  - and *L* becomes *(a,b,d,e,f,g)*
  - index of *d,e,f,* and *g* decrease by *1*

Linear List Operations

- Add(theIndex, theElement)
  - add an element so that the new element has a specified index
  - *L = (a,b,c,d,e,f,g)*
  - *add(0,h)* => *L = (h,a,b,c,d,e,f,g)*
  - index of *a,b,c,d,e,f,* and *g* increase by *1*
  - *add(10,h)* => error
  - *add(-6,h)* => error

Linear List Array Representation

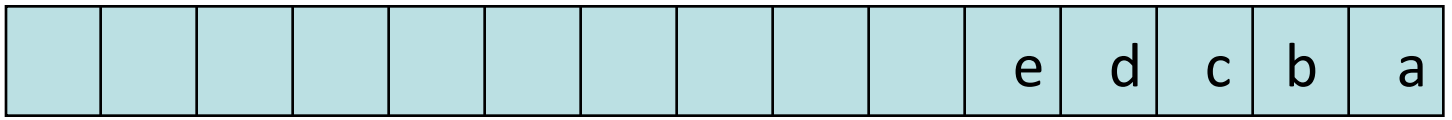
use a one-dimensional array element[]



L = (a, b, c, d, e)

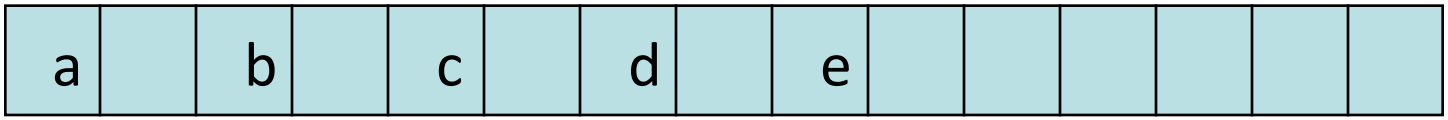
Store element i of list in element[i].

- Right to left mapping

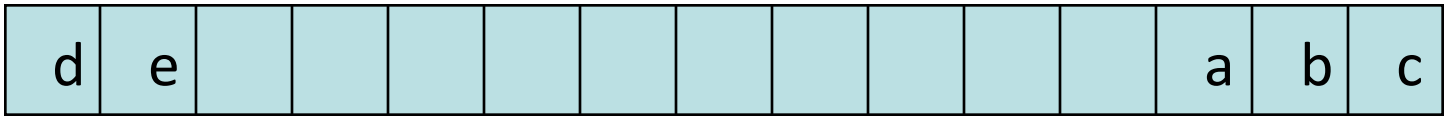


Linear List Array Representation

- Mapping That Skips Every Other Position

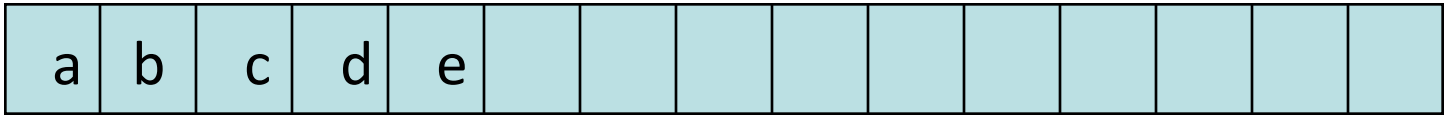


- Wrap Around Mapping



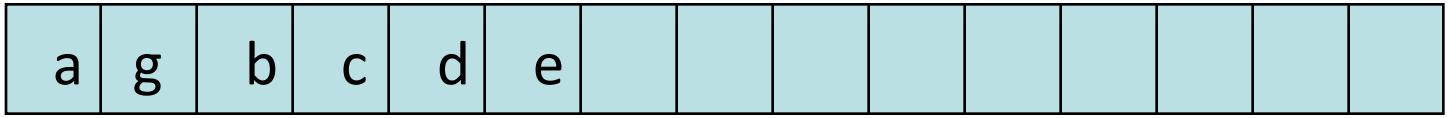
Add/Remove An Element

size = 5



add(1,g)

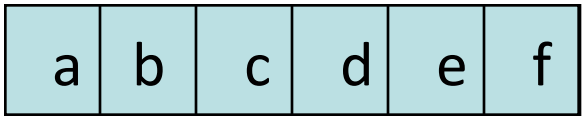
size = 6



Increasing Array Length

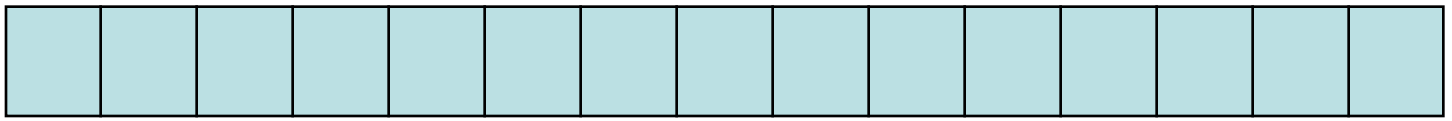
- Don't know how many elements will be in list.
- Must pick an initial length and dynamically increase as needed.

Length of array element[] is 6.



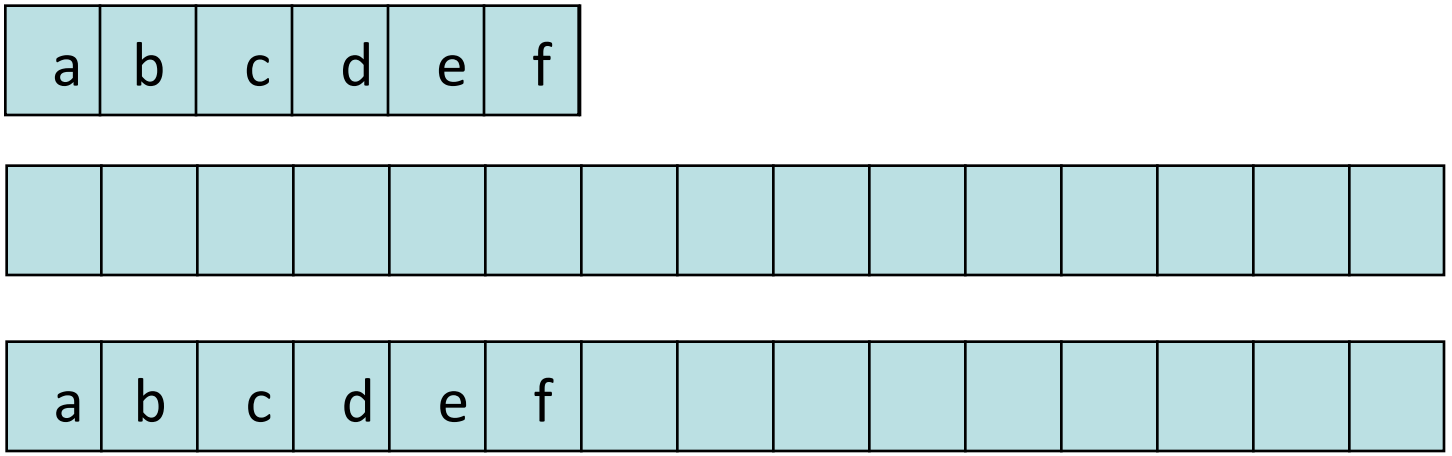
First create a new and larger array

newArray = new Object[15];

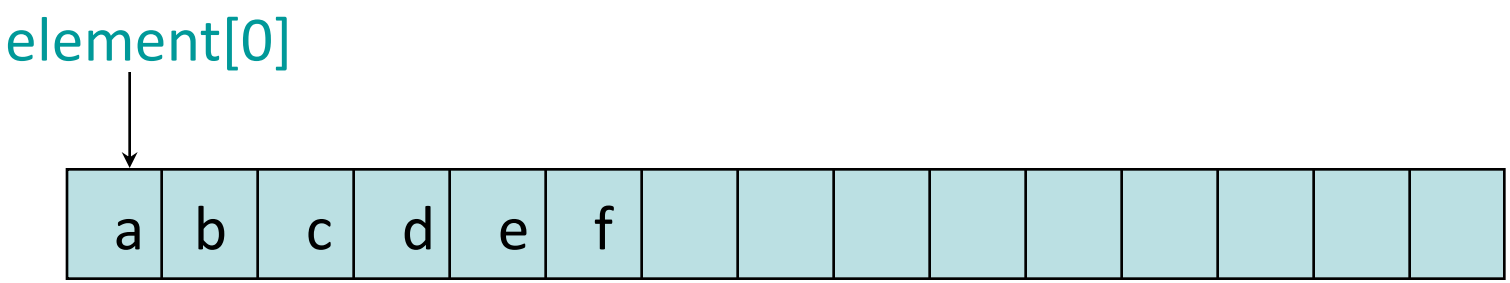


Increasing Array Length

Now copy elements from old array to new one.



Finally, rename new array.



How Big Should The New Array Be?

At least 1 more than current array length.

Cost of increasing array length is

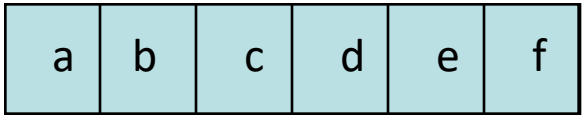
Theta(new length)

Cost of n add operations done on an initially empty linear list increases by

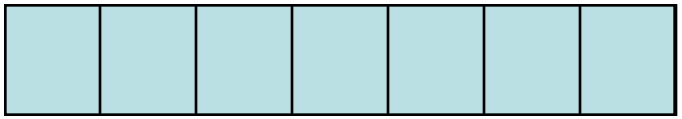
Theta(n^2)

Space Complexity

element[6]



newArray = new char[7];

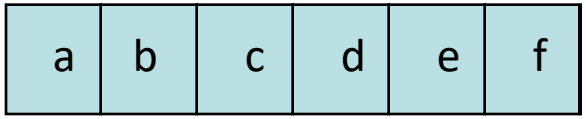


space needed = 2 * newLength – 1

= 2 * maxListSize – 1

Array Doubling

Double the array length.



newArray = new char[12];

Time for n adds goes up by Theta(n).

Space needed = 1.5*newLength.

Space needed <= 3*maxListSize – 3

How Big Should The New Array Be?

Resizing by any constant factor

new length = c * old length

increases the cost of n adds by Theta(n).

Resizing by an additive constant increases the cost of n add operations by Theta(n^2).

How Big Should The New Array Be?

Resizing by any constant factor

new length = c * old length

requires at most (1+c) * (maxListSize -1) space.

Resizing by an additive constant c requires

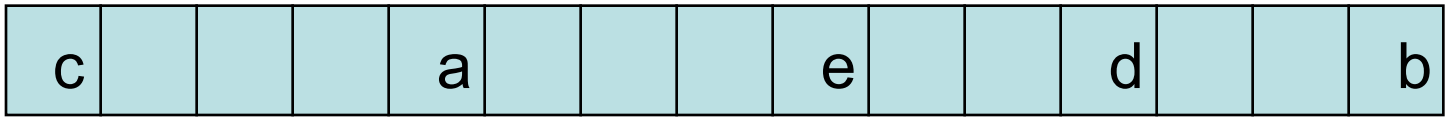
at most (maxListSize – 1) + (maxListSize – 1 + c)

= 2 * (maxListSize – 1) + c space.

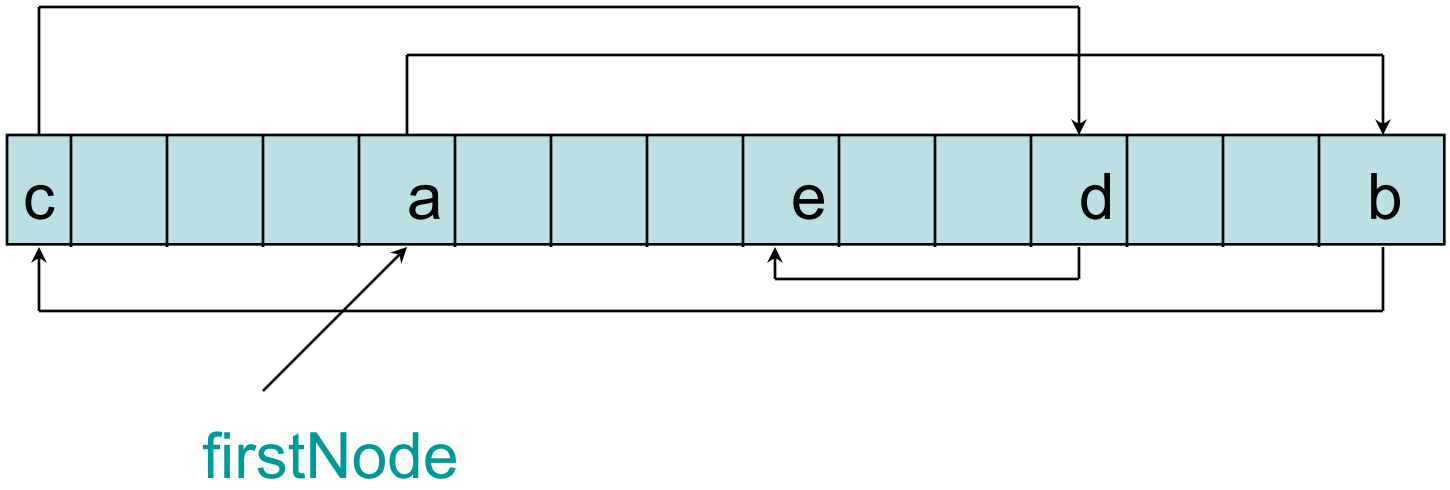
Linked Representation

- list elements are stored, in memory, in an arbitrary order
- explicit information (called a link) is used to go from one element to the next

A linked representation uses an arbitrary layout.



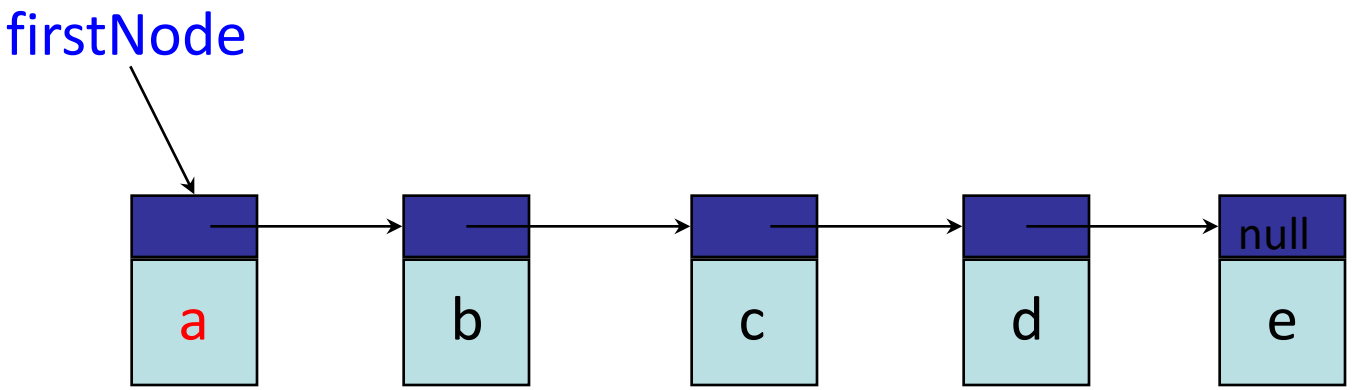
Linked Representation



- pointer (or link) in e is null

use a variable firstNode to get to the first element a

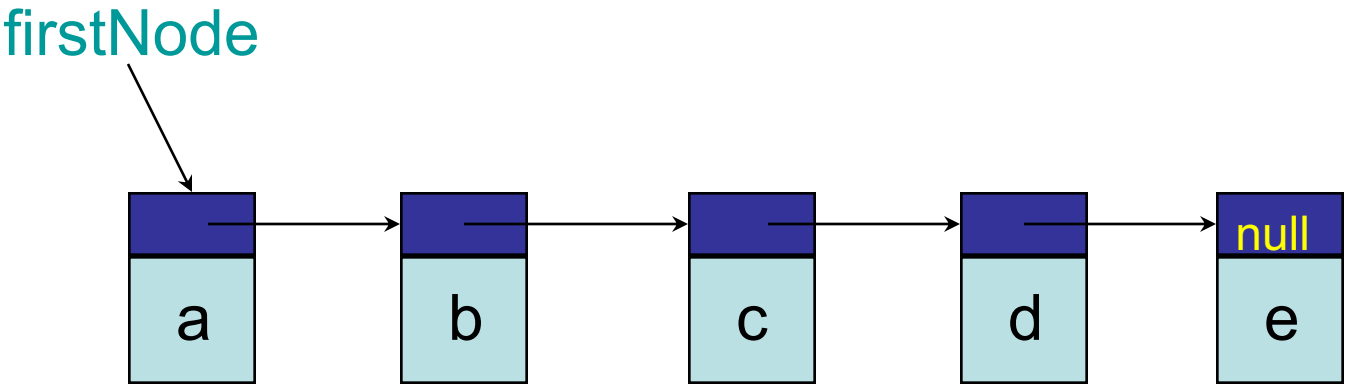
Normal Way To Draw A Linked List



- link or pointer field of node

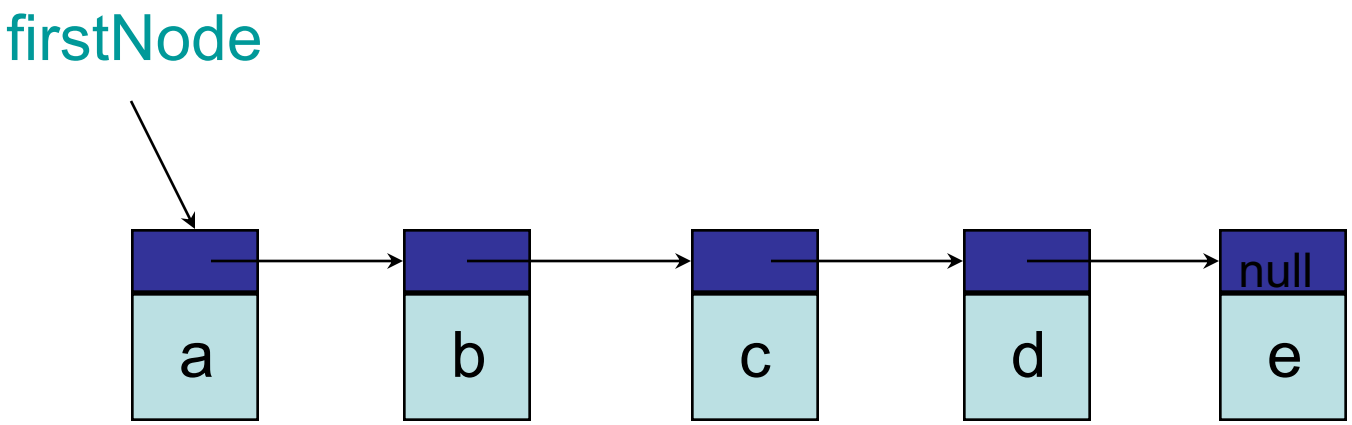
data field of node

Chain



- A chain is a linked list in which each node represents one element.
- There is a link or pointer from one element to the next.
- The last node has a null pointer.

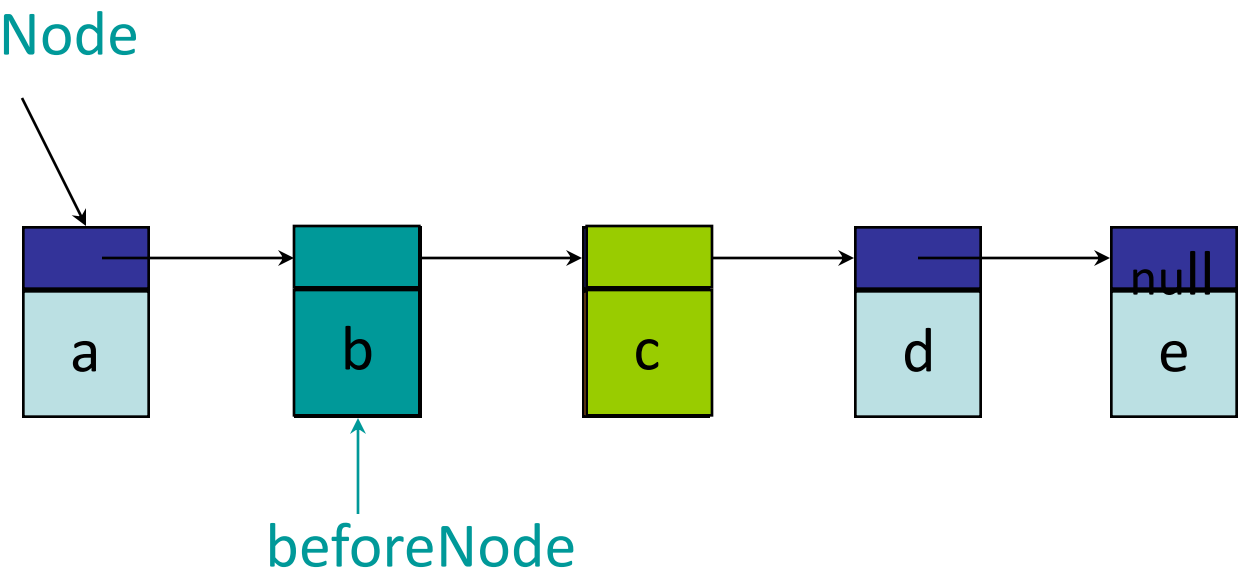
List Operations: get()



- checkIndex(1);
- desiredNode = firstNode.nextreturn desiredNode.element;

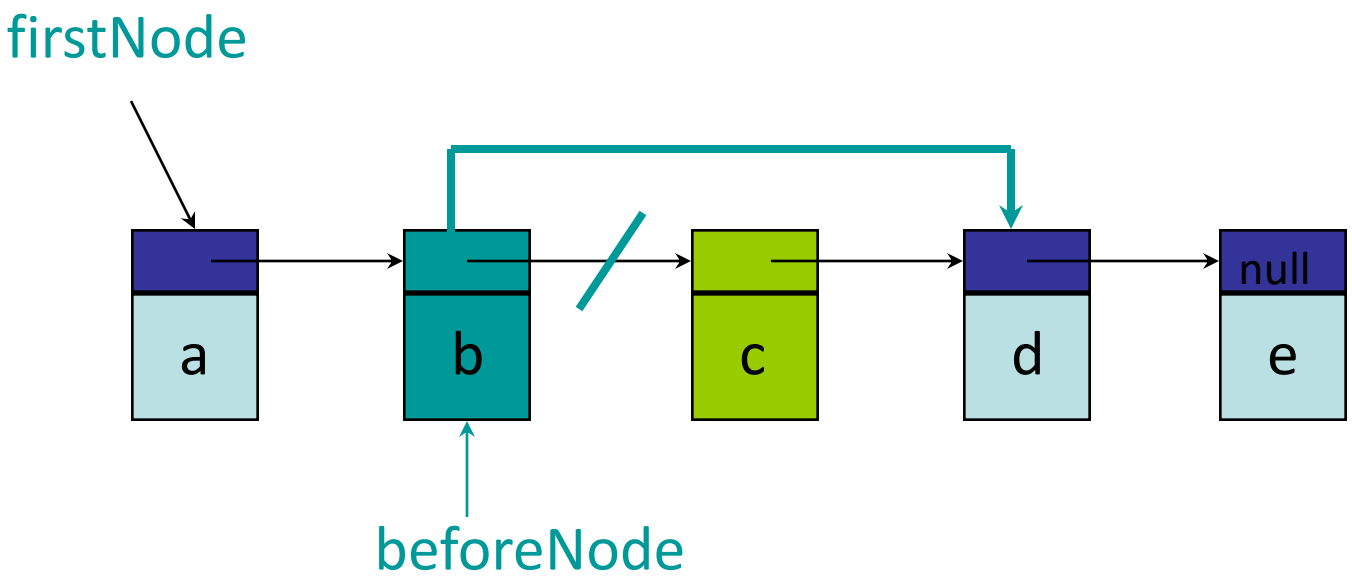
List Operations: Remove

Remove (2)



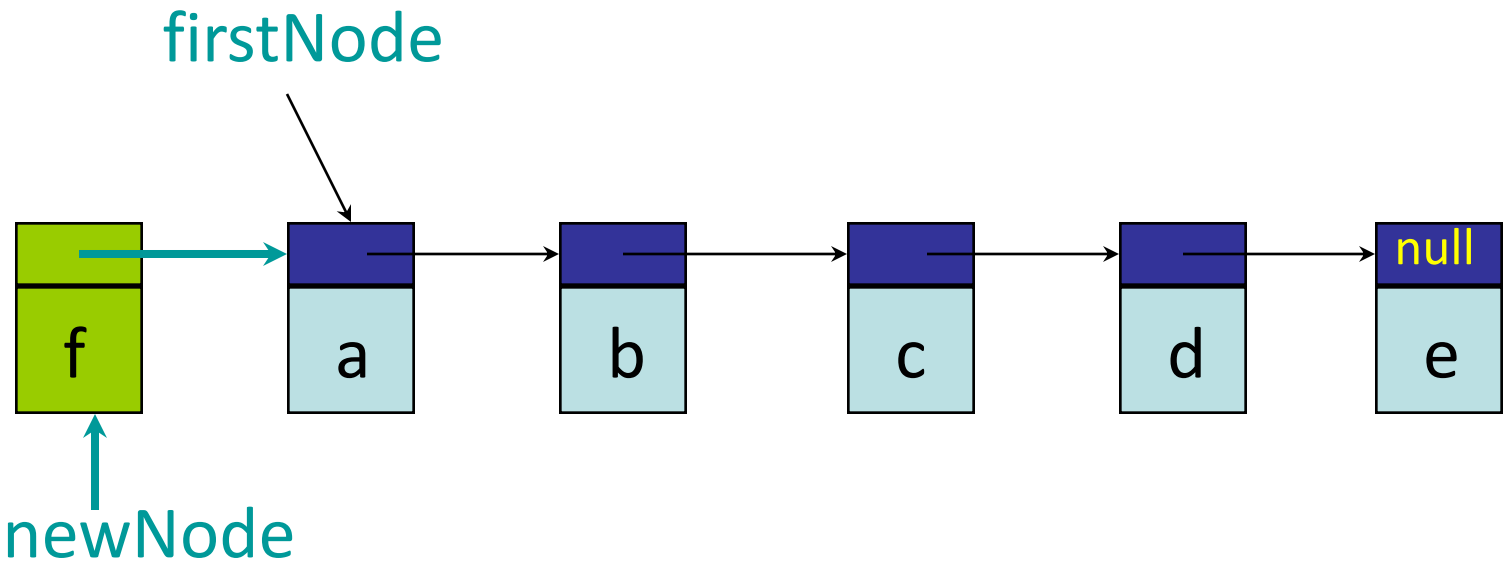
- first get to node just before node to be removed
- beforeNode = firstNode.next;

List Operations: Remove



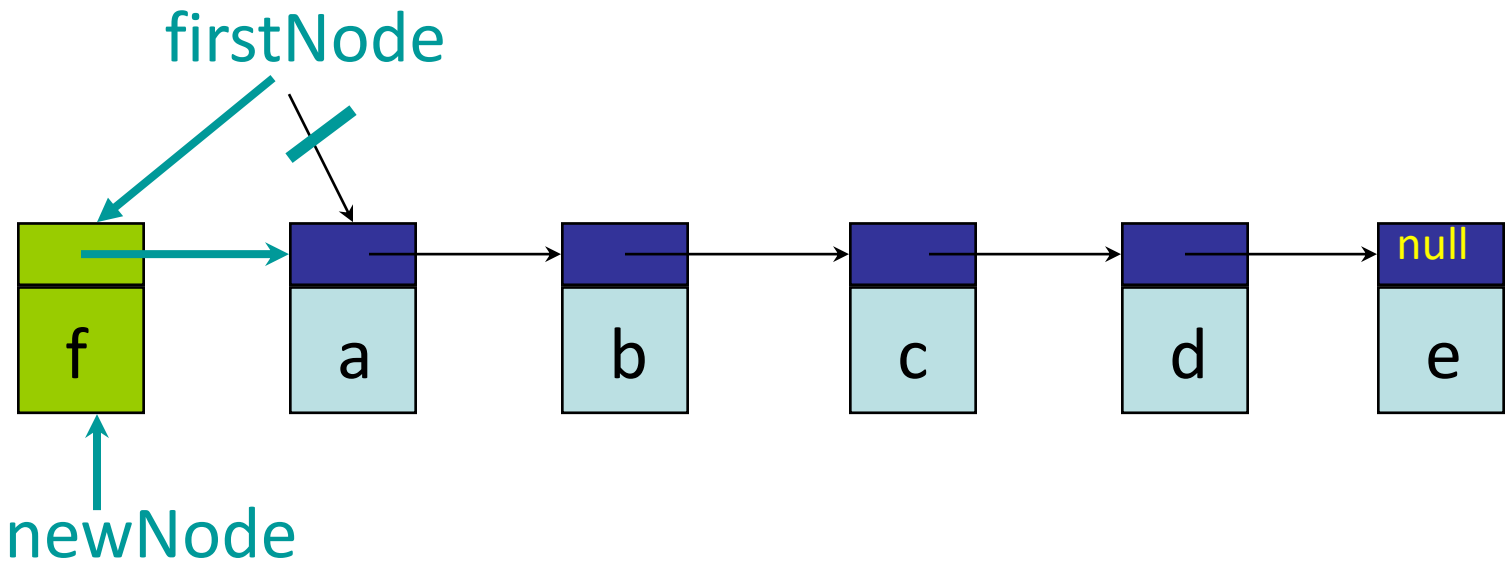
- now change pointer in beforeNode
-

List Operations: Add



Step 1: get a node, set its data and link fields

List Operations: Add



Step 2: update firstNode

```
firstNode = newNode;
```

Performance

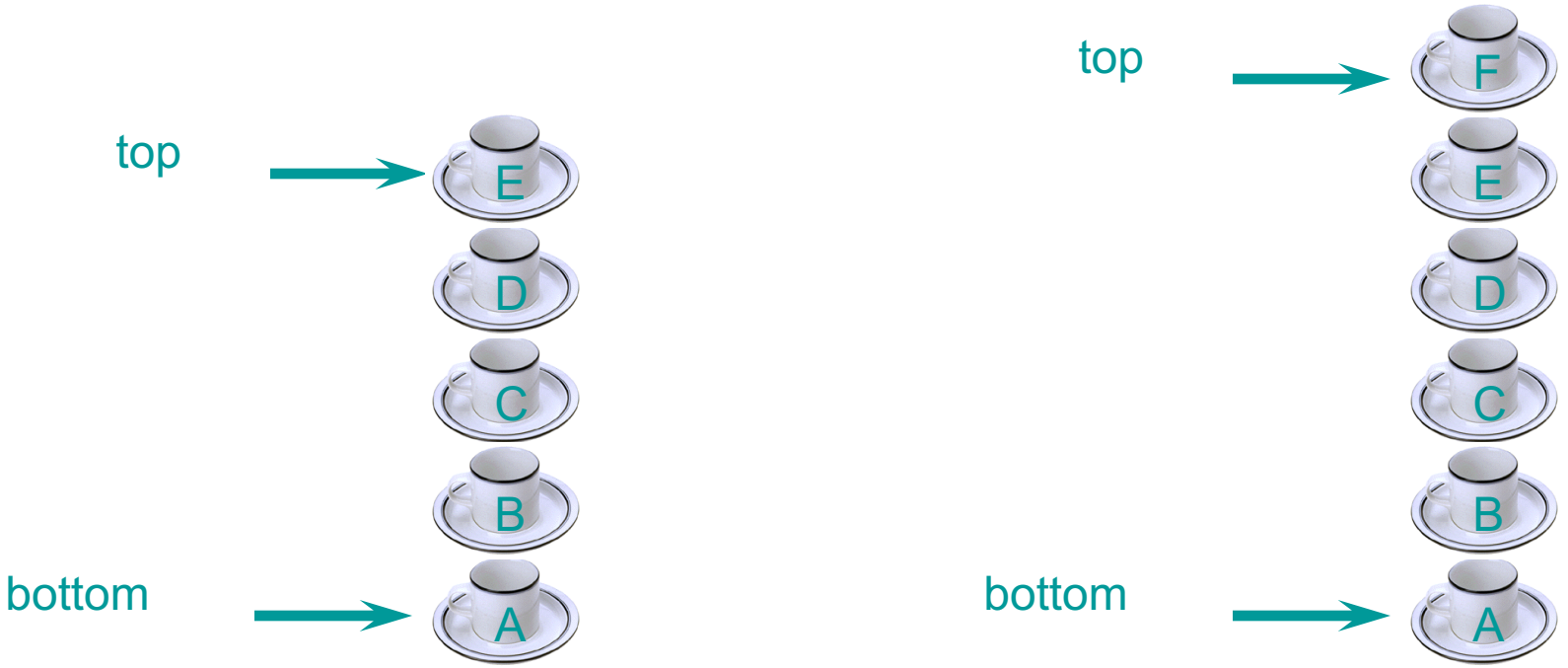
- 40,000 operations of each type

Operation	FastArrayLinearList	Chain
get	5.6ms	157sec
best-case adds	31.2ms	304ms
average adds	5.8sec	115sec
worst-case adds	11.8sec	157sec
best-case removes	8.6ms	13.2ms
average removes	5.8sec	149sec
worst-case removes	11.7sec	157sec

Stacks

- Linear list.
- One end is called top.
- Other end is called bottom.
- Additions to and removals from the top end only.

Stack Of Cups



- Add a cup to the stack.
- Remove a cup from new stack.
- A stack is a LIFO list.

The Interface Stack

- empty();
- peek();
- push(theObject);
- pop();

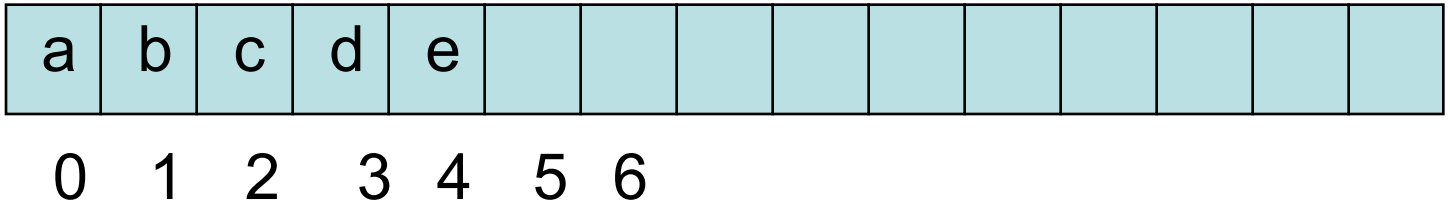
Stack Applications

- Parentheses matching.
- Towers of Hanoi.
- Switchbox routing.
- Method invocation and return.
- Try-catch-throw implementation.

Derive From A Linear List

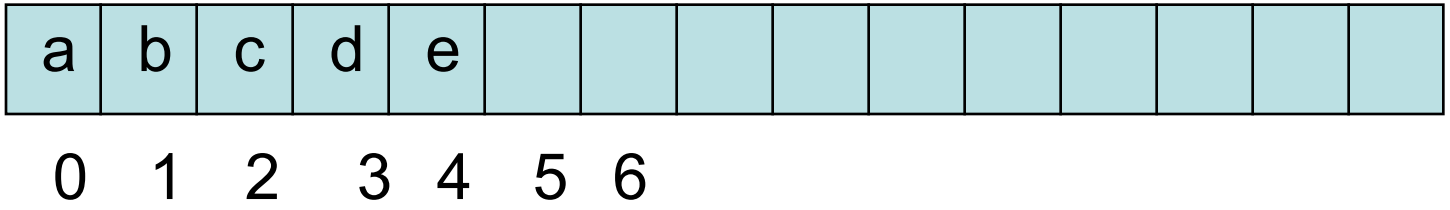
- ArrayLinearList
- Chain

Derive From ArrayLinearList



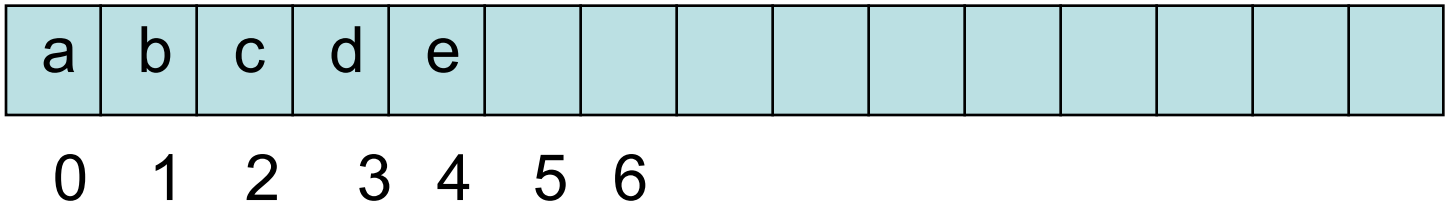
- stack top is either left end or right end of linear list
- empty() => isEmpty()
  - O(1) time
- peek() => get(0) or get(size() - 1)
  - O(1) time

Derive From ArrayLinearList



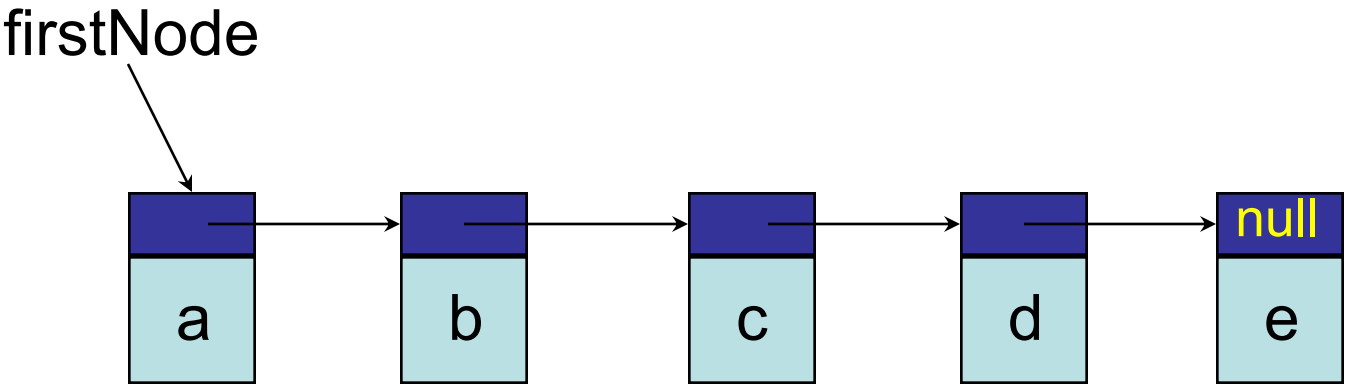
- when top is left end of linear list
  - push(theObject) => add(0, theObject)
  - O(size) time
  - pop() => remove(0)
  - O(size) time

Derive From ArrayLinearList



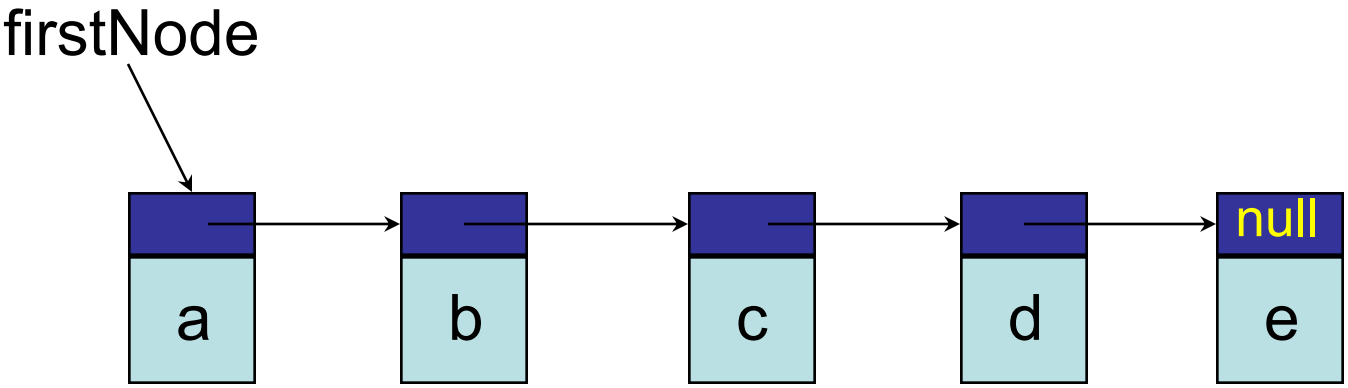
- when top is right end of linear list
  - push(theObject) => add(size(), theObject)
  - O(1) time
  - pop() => remove(size()-1)
  - O(1) time
- use right end of list as top of stack

Derive From Chain



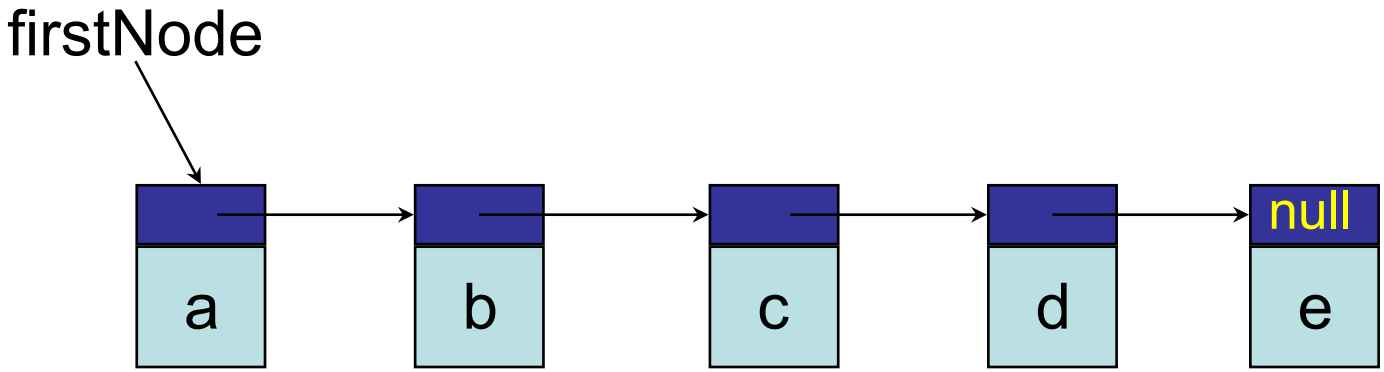
- stack top is either left end or right end of linear list
- empty() => isEmpty()
  - O(1) time

Derive From Chain



- when top is left end of linear list
  - peek() => get(0)
  - O(1) time
  - push(theObject) => add(0, theObject)
  - O(1) time
  - pop() => remove(0)
  - O(1) time

Derive From Chain



- when top is right end of linear list
  - peek() => get(size() - 1)
  - O(size) time
  - push(theObject) => add(size(), theObject)
  - O(size) time
  - pop() => remove(size()-1)
  - O(size) time
- use left end of list as top of stack

Queues

- Linear list.
- One end is called front.
- Other end is called rear.
- Additions are done at the rear only.
- Removals are made from the front only.

Bus Stop Queue

front rear

Bus Stop Queue

front

rear

Bus Stop Queue

front rear

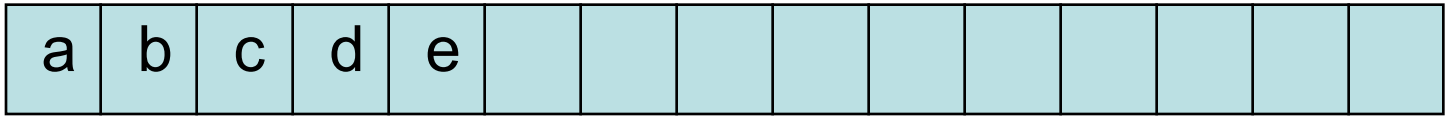
Bus Stop Queue

front rear

The Interface Queue

- Empty();
- Element();
- getRearElement();
- put (theObject);
- remove();

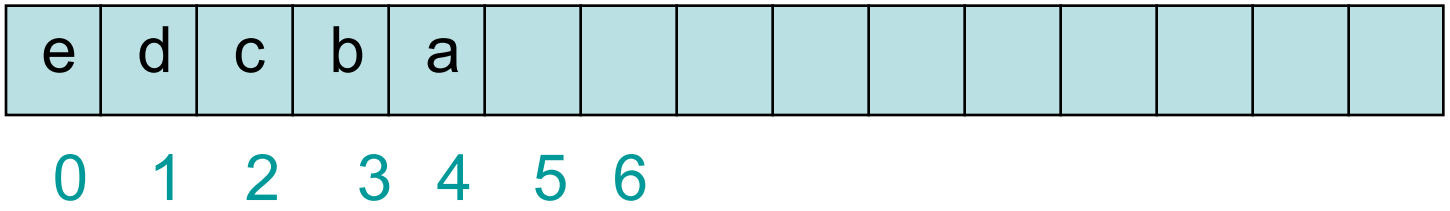
Derive From ArrayLinearList



when front is left end of list and rear is right end

- Queue.isEmpty() => super.isEmpty()
  - O(1) time
- getFrontElement() => get(0)
  - O(1) time
- getRearElement() => get(size() - 1)
  - O(1) time
- put(theObject) => add(size(), theObject)
  - O(1) time
- remove() => remove(0)
  - O(size) time

Derive From ArrayLinearList



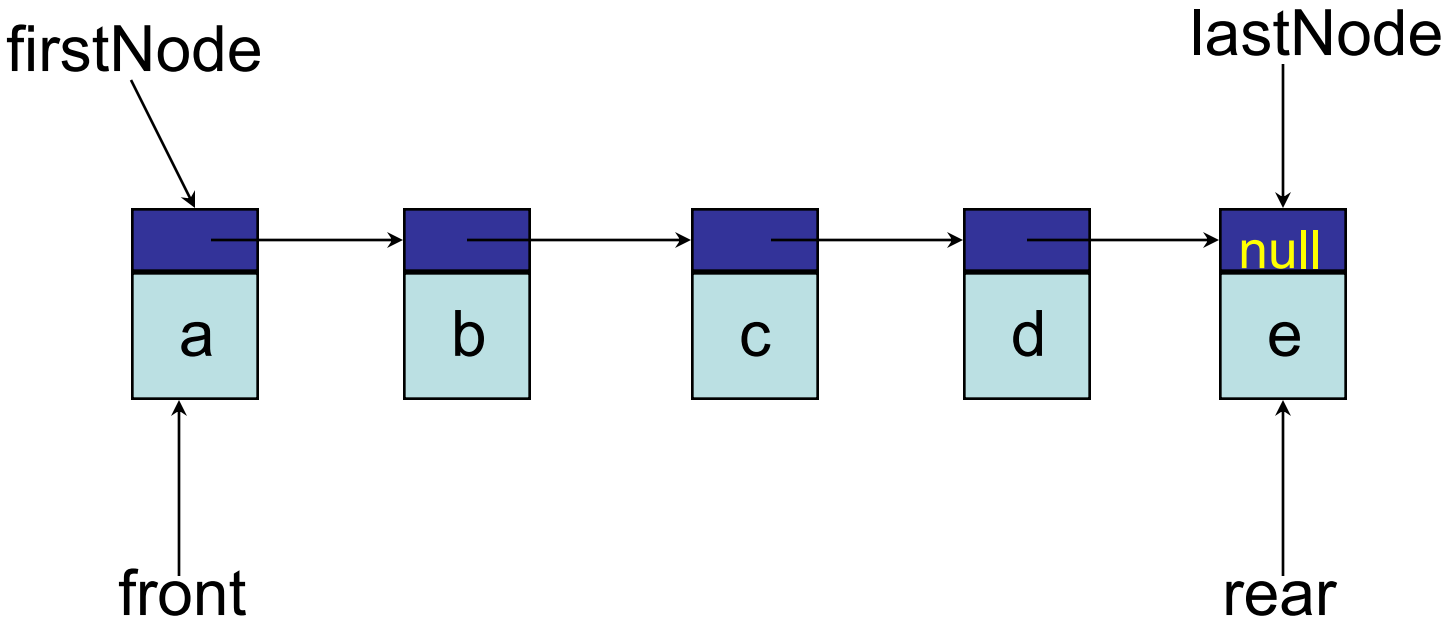
– when rear is left end of list and front is right end

- Queue.isEmpty() => super.isEmpty()
  - O(1) time
- getFrontElement() => get(size() - 1)
  - O(1) time
- getRearElement() => get(0)
  - O(1) time
- put(theObject) => add(0, theObject)
  - O(size) time
- remove() => remove(size() - 1)
  - O(1) time

Derive From ArrayLinearList

– to perform each operation in O(1) time (excluding array doubling), we need a customized array representation.

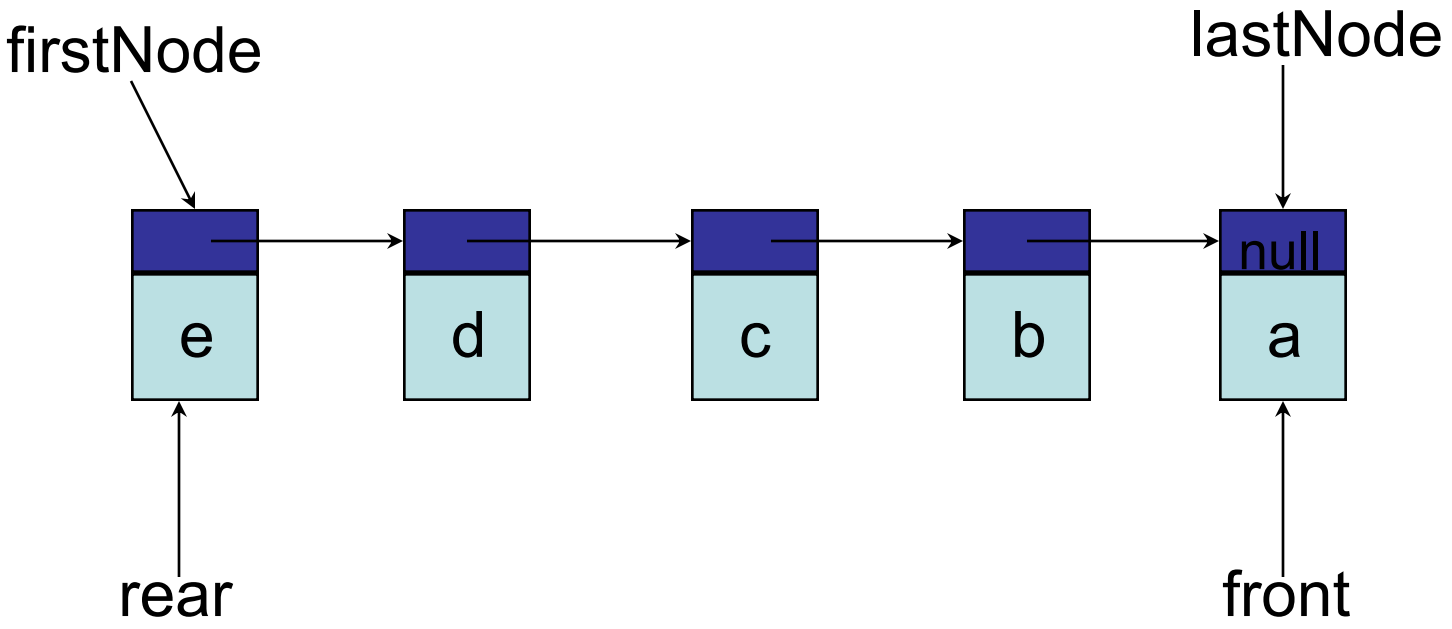
Derive From ExtendedChain



when front is left end of list and rear is right end

- Queue.isEmpty() => super.isEmpty()
  - O(1) time
- getFrontElement() => get(0)
  - O(1) time

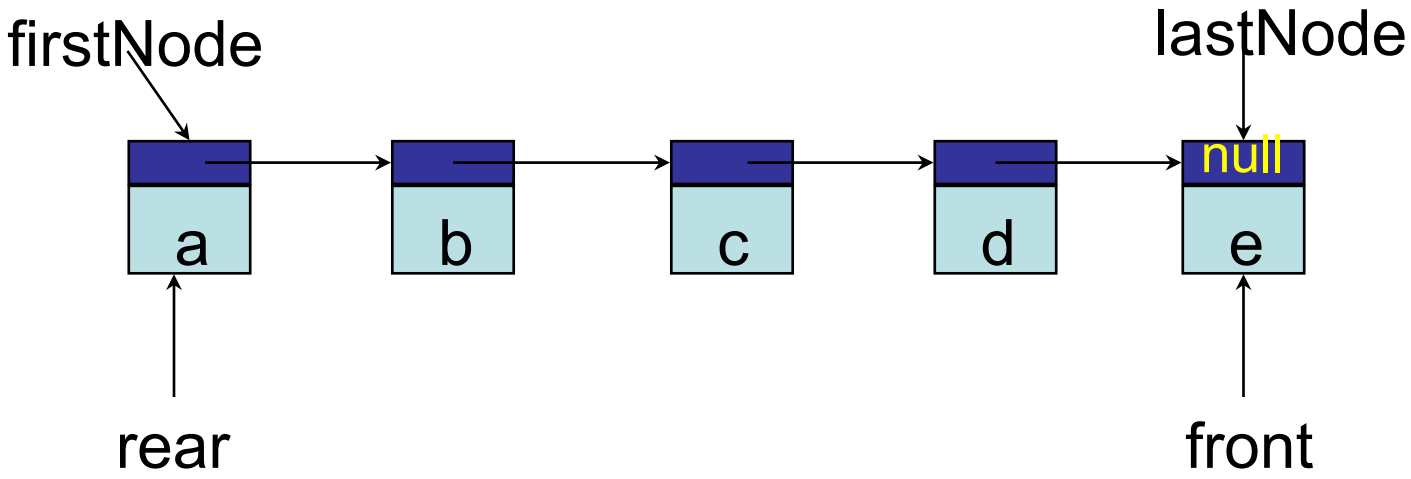
Derive From ExtendedChain



when front is right end of list and rear is left end

- Queue.isEmpty() => super.isEmpty()
  - O(1) time
- getFrontElement() => getLast()
  - O(1) time

Derive From ExtendedChain



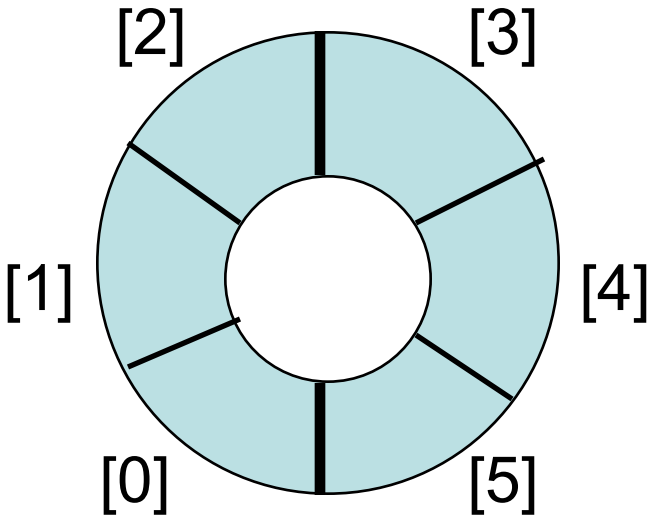
- getRearElement() => get(0)
  - O(1) time
- put(theObject) => add(0, theObject)
  - O(1) time
- remove() => remove(size-1)
  - O(size) time

Custom Array Queue

- Use a 1D array queue.

queue[]

- Circular view of array.



Thank You