POSTGRADUATE DEPARTMENT OF COMPUTER APPLICATIONS,
GOVERNMENT ARTS COLLEGE(AUTONOMOUS),
COIMBATORE 641018.

DATA STRUCTURES AND ALGORITHMS

The contents in this E material are from

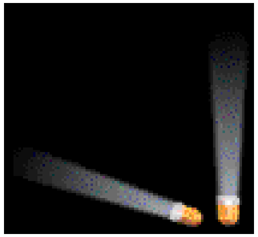
Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
“Fundamentals of Data Structures in C”,
Computer Science Press, 1992.

UNIT 2

FACULTY

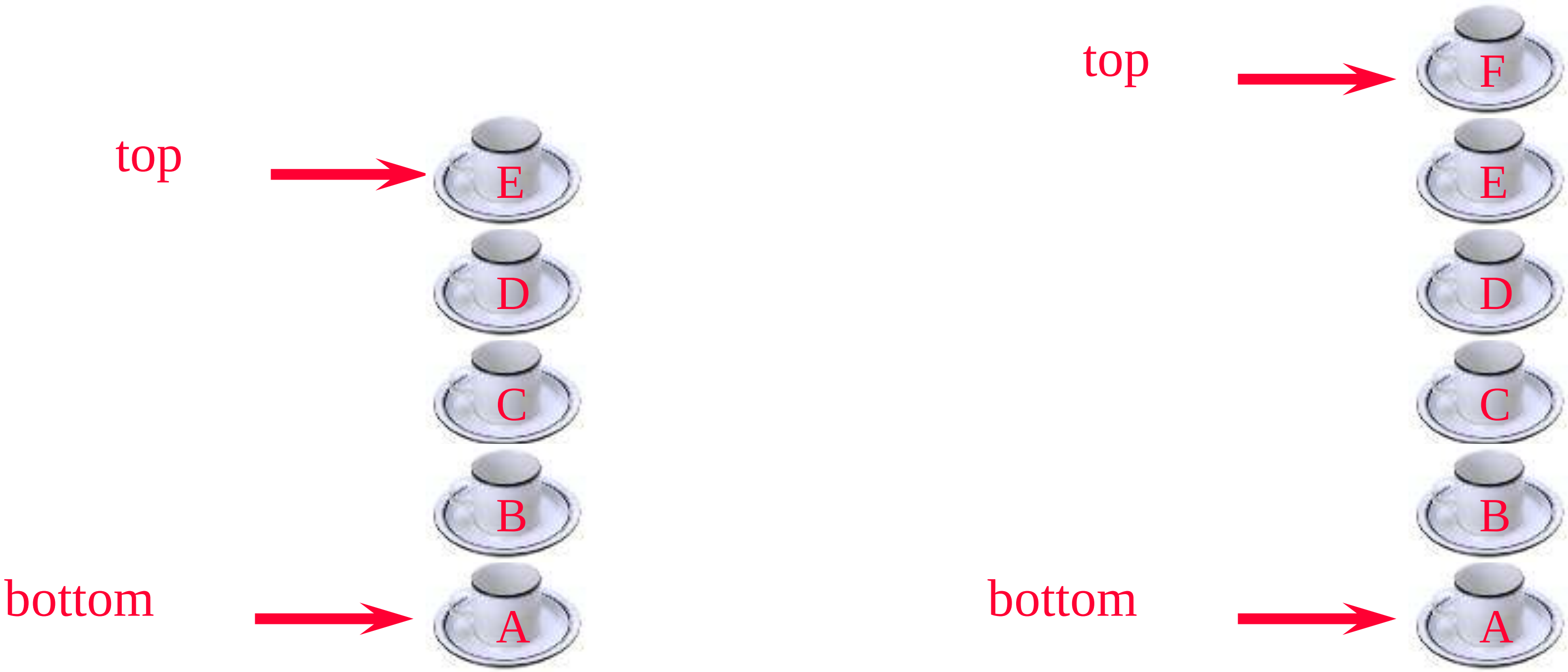
Dr.R.A.ROSELINE M.Sc.M.Phil.,Ph.D,
Associate Professor and Head,
Postgraduate Department of Computer Applications,
Government Arts College(Autonomous),
Coimbatore 641018.

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.

Parentheses Matching

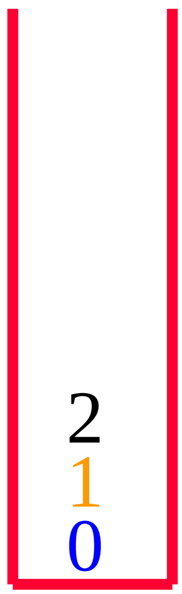
- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)
  - Output pairs (u,v) such that the left parenthesis at position u is matched with the right parenthesis at v.
    - (2,6) (1,13) (15,19) (21,25) (27,31) (0,32) (34,38)
- (a+b))*((c+d)
  - (0,4)
  - right parenthesis at 5 has no matching left parenthesis
  - (8,12)
  - left parenthesis at 7 has no matching right parenthesis

Parentheses Matching

- scan expression from left to right
- when a left parenthesis is encountered, add its position to the stack
- when a right parenthesis is encountered, remove matching position from stack

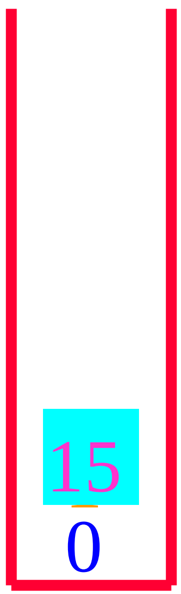
Example

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)



Example

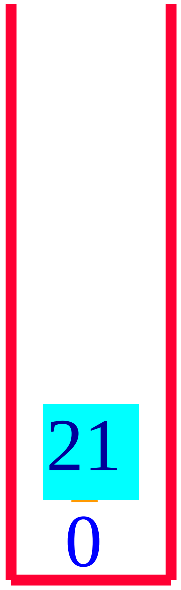
- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)



(2,6) (1,13)

Example

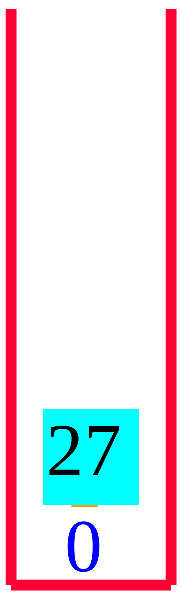
- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)



(2,6) (1,13) (15,19)

Example

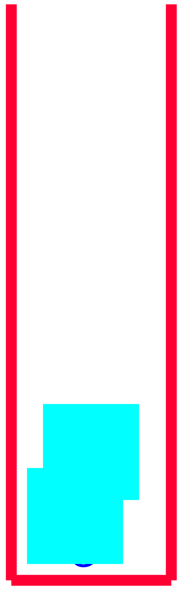
- ((((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)



(2,6) (1,13) (15,19) (21,25)

Example

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)



(2,6) (1,13) (15,19) (21,25)(27,31) (0,32)

- and so on

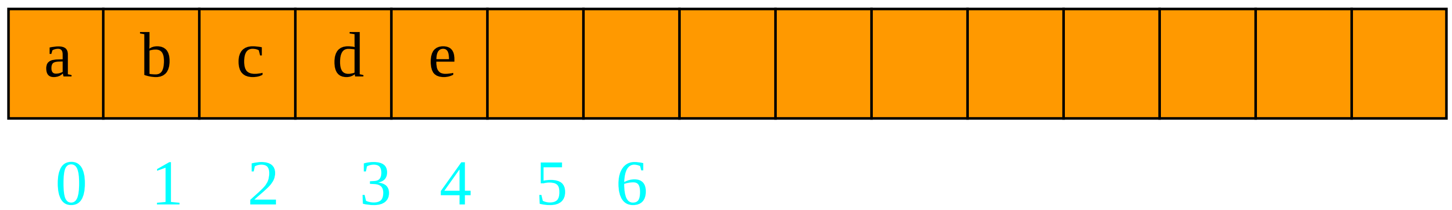
Stacks

- Standard operations:
  - IsEmpty … return true iff stack is empty
  - IsFull … return true iff stack has no remaining capacity
  - Top … return top element of stack
  - Push … add an element to the top of the stack
  - Pop … delete the top element of the stack

Stacks

- Use a 1D array to represent a stack.
- Stack elements are stored in stack[0] through stack[top].

Stacks



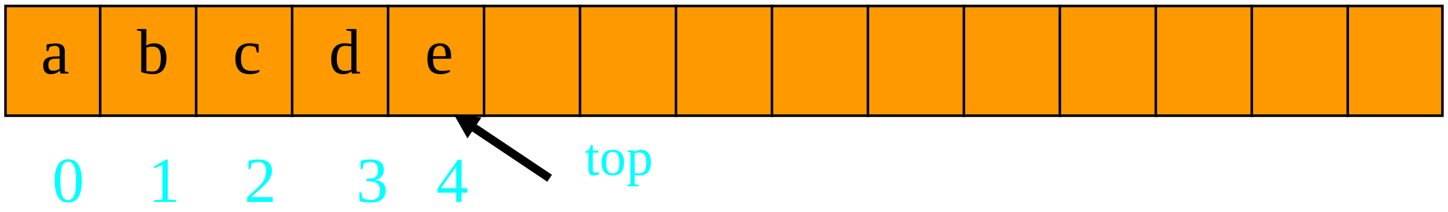
- stack top is at element e
- IsEmpty() => check whether top >= 0
  - O(1) time
- IsFull() => check whether top == capacity – 1
  - O(1) time
- Top() => If not empty return stack[top]
  - O(1) time

Derive From arrayList

a	b	c	d	e										
0	1	2	3	4	5	6								

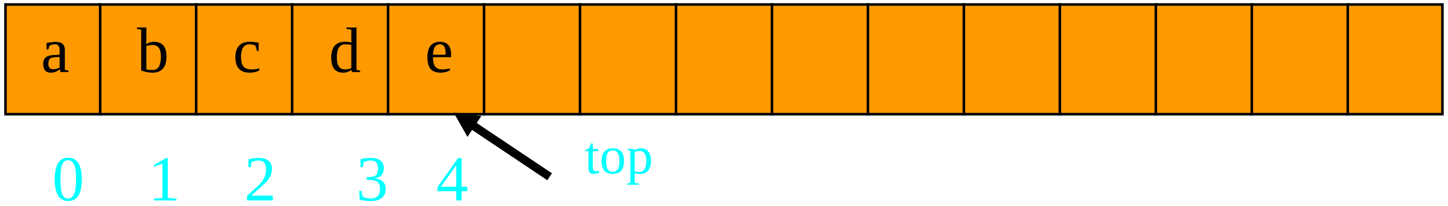
- Push(theElement) => if full then either error or increase capacity and then add at stack[top+1]
- Suppose we increase capacity when full
- O(capacity) time when full; otherwise O(1)
- Pop() => if not empty, delete from stack[top]
- O(1) time

Push




```
void push(element item)
{/* add an item to the global stack */
   if (top >= MAX_STACK_SIZE - 1)
       StackFull();
   /* add at stack top */
   stack[++top] = item;
}
```

Pop




```
element pop()
{
    if (top == -1)
        return StackEmpty();
    return stack[top--];
}
```

StackFull()

```
void StackFull()
{
    fprintf(stderr, “Stack is full, cannot add
                                    element.”);
    exit(EXIT_FAILURE);
}
```

StackFull()/Dynamic Array

- Use a variable called *capacity* in place of MAX_STACK_SIZE
- Initialize this variable to (say) 1
- When stack is full, double the capacity using REALLOC
- This is called *array doubling*

StackFull()/Dynamic Array

```
void StackFull()
{
    REALLOC(stack, 2*capacity*sizeof(*stack);
    capacity *= 2;
}
```

Complexity Of Array Doubling

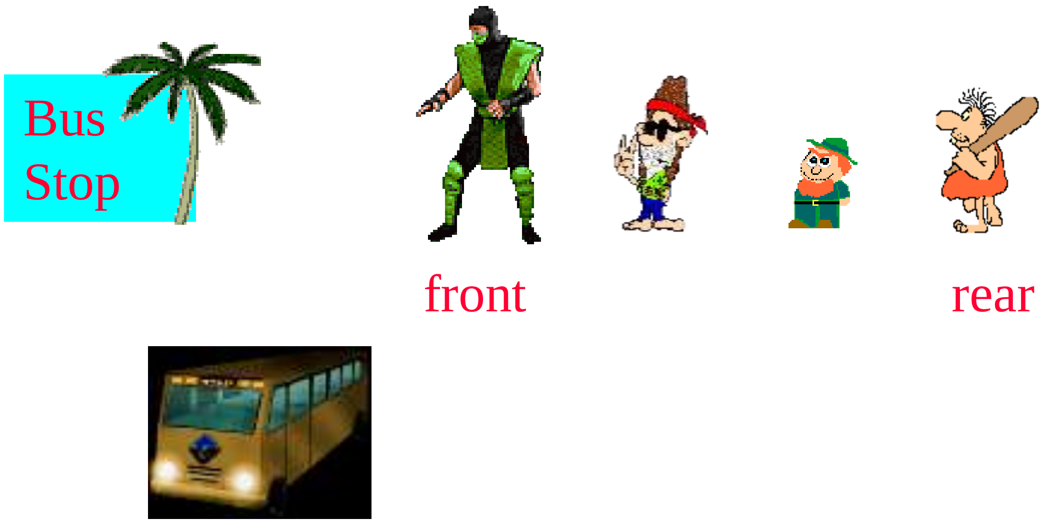
- Let final value of capacity be 2^k
- Number of pushes is at least $2^{k-1}+ 1$
- Total time spent on array doubling is $\Sigma_{1<=i=k}2^i$
- This is $O(2^k)$
- So, although the time for an individual push is O(capacity), the time for all n pushes remains O(n)!

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

Bus Stop Queue



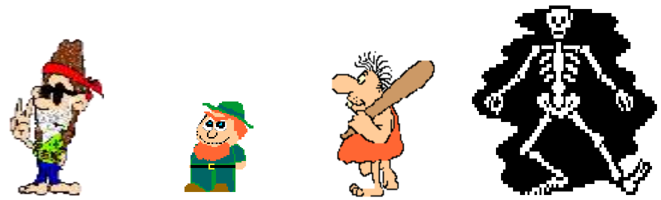
Bus Stop Queue

Bus
Stop

front

rear

Bus Stop Queue

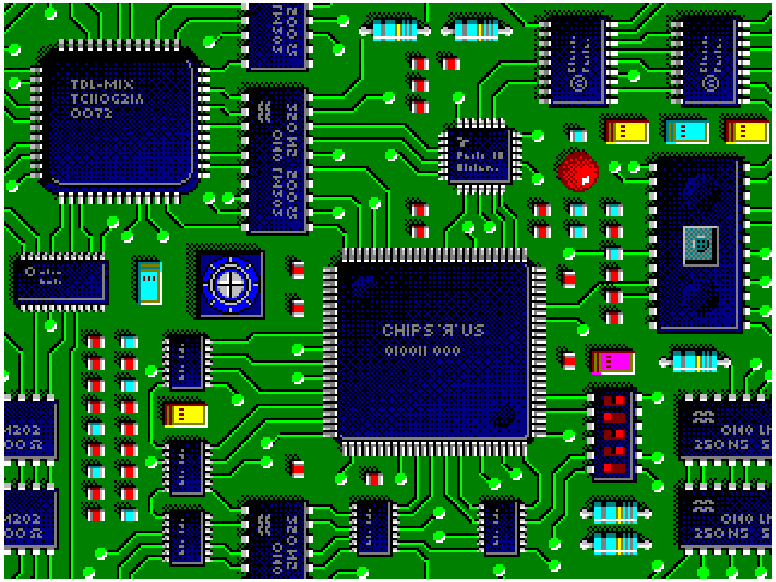


front rear

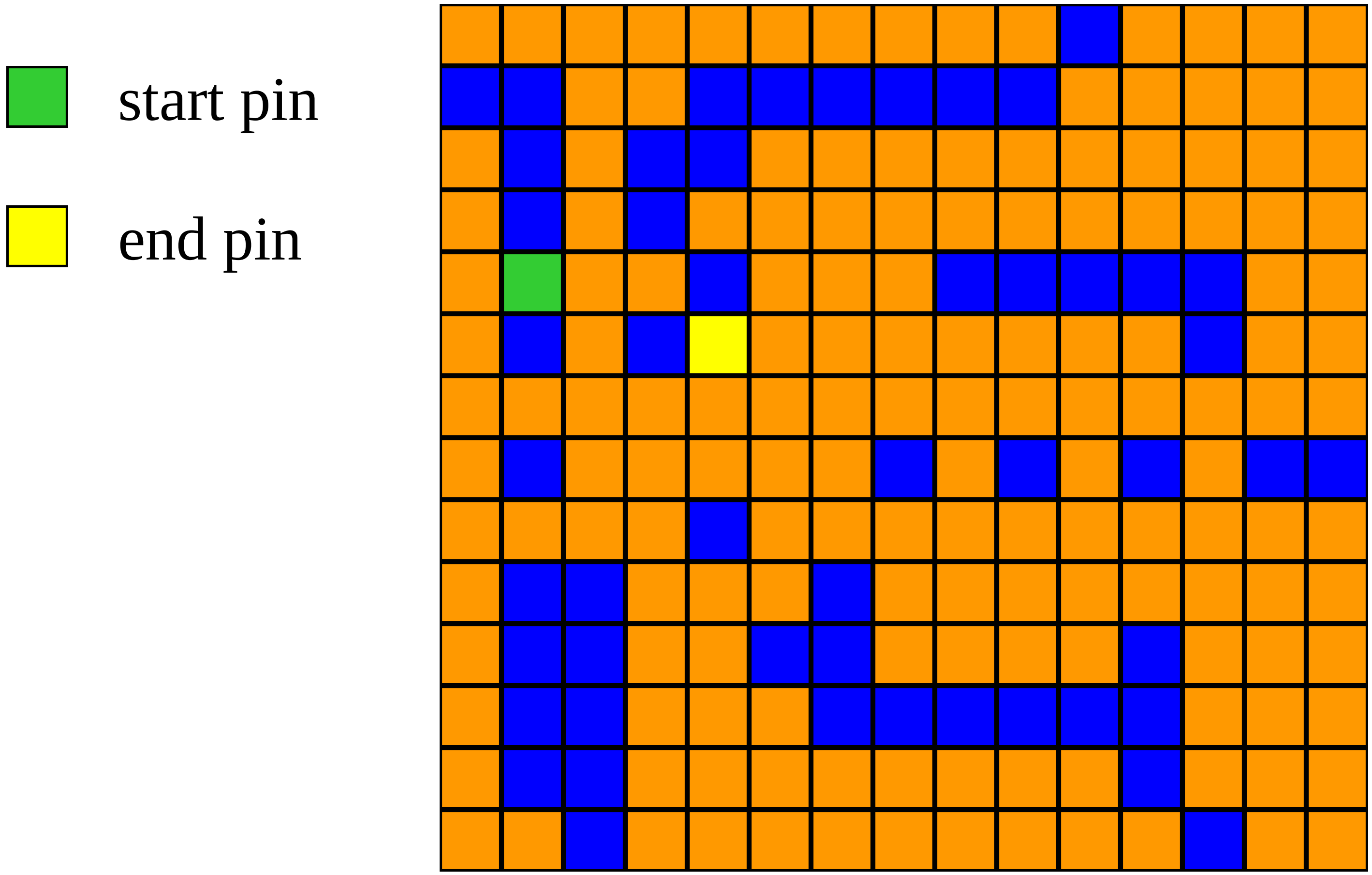
Revisit Of Stack Applications

- Applications in which the stack cannot be replaced with a queue.
  - Parentheses matching.
  - Towers of Hanoi.
  - Switchbox routing.
  - Method invocation and return.
  - Try-catch-throw implementation.
- Application in which the stack may be replaced with a queue.
  - Rat in a maze.
    - Results in finding shortest path to exit.

Wire Routing

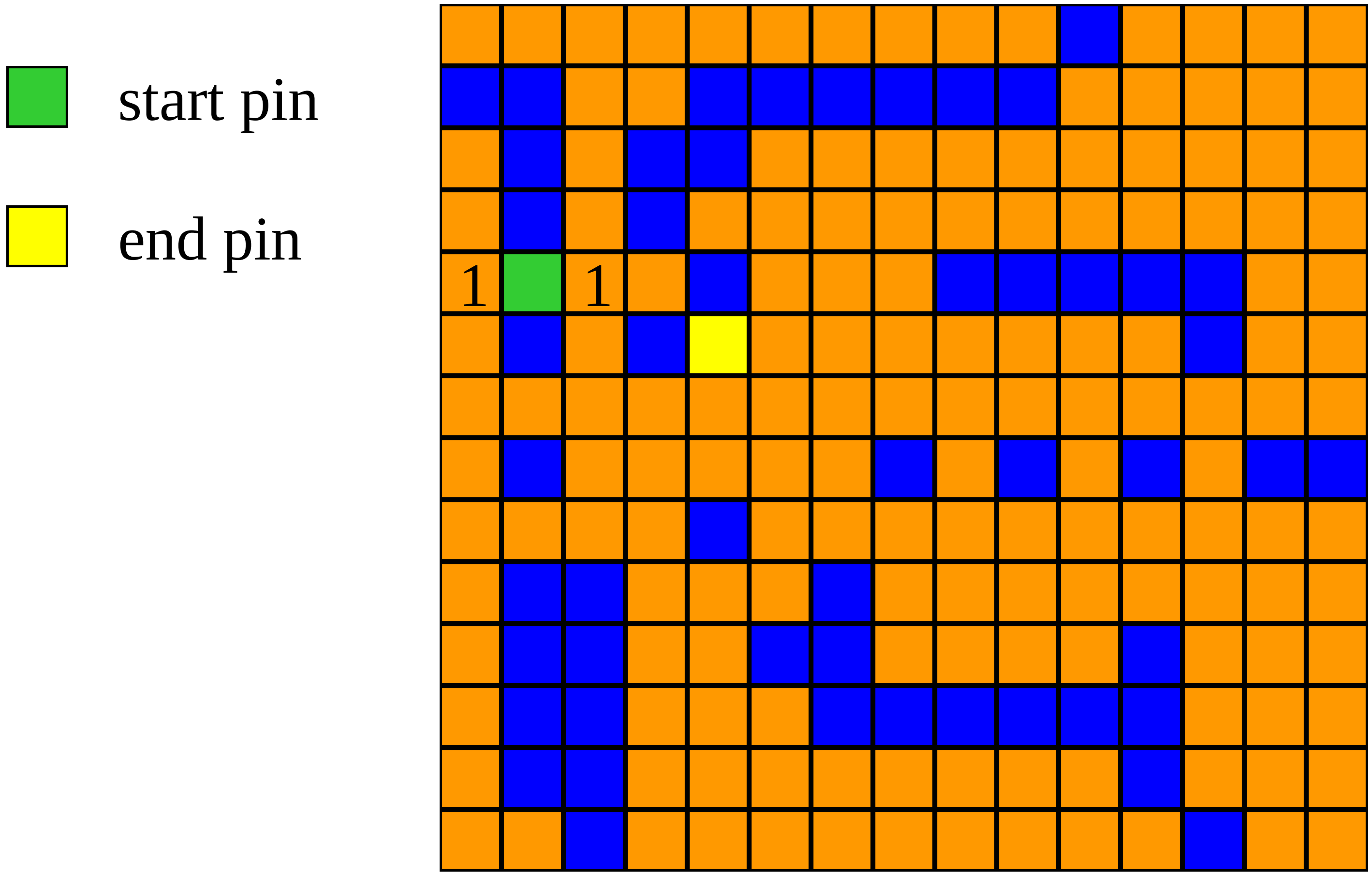


Lee's Wire Router



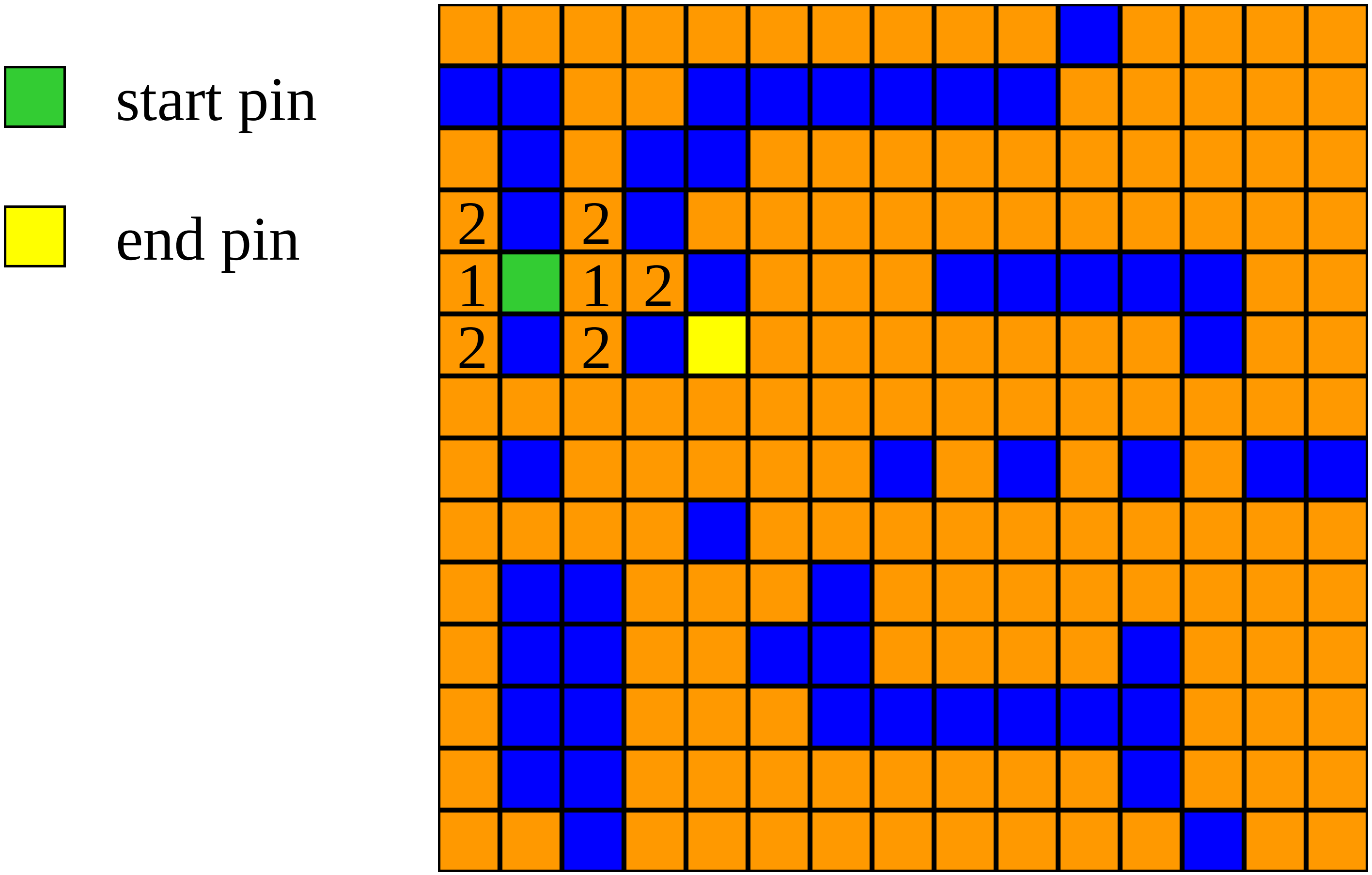
Label all reachable squares 1 unit from start.

Lee's Wire Router



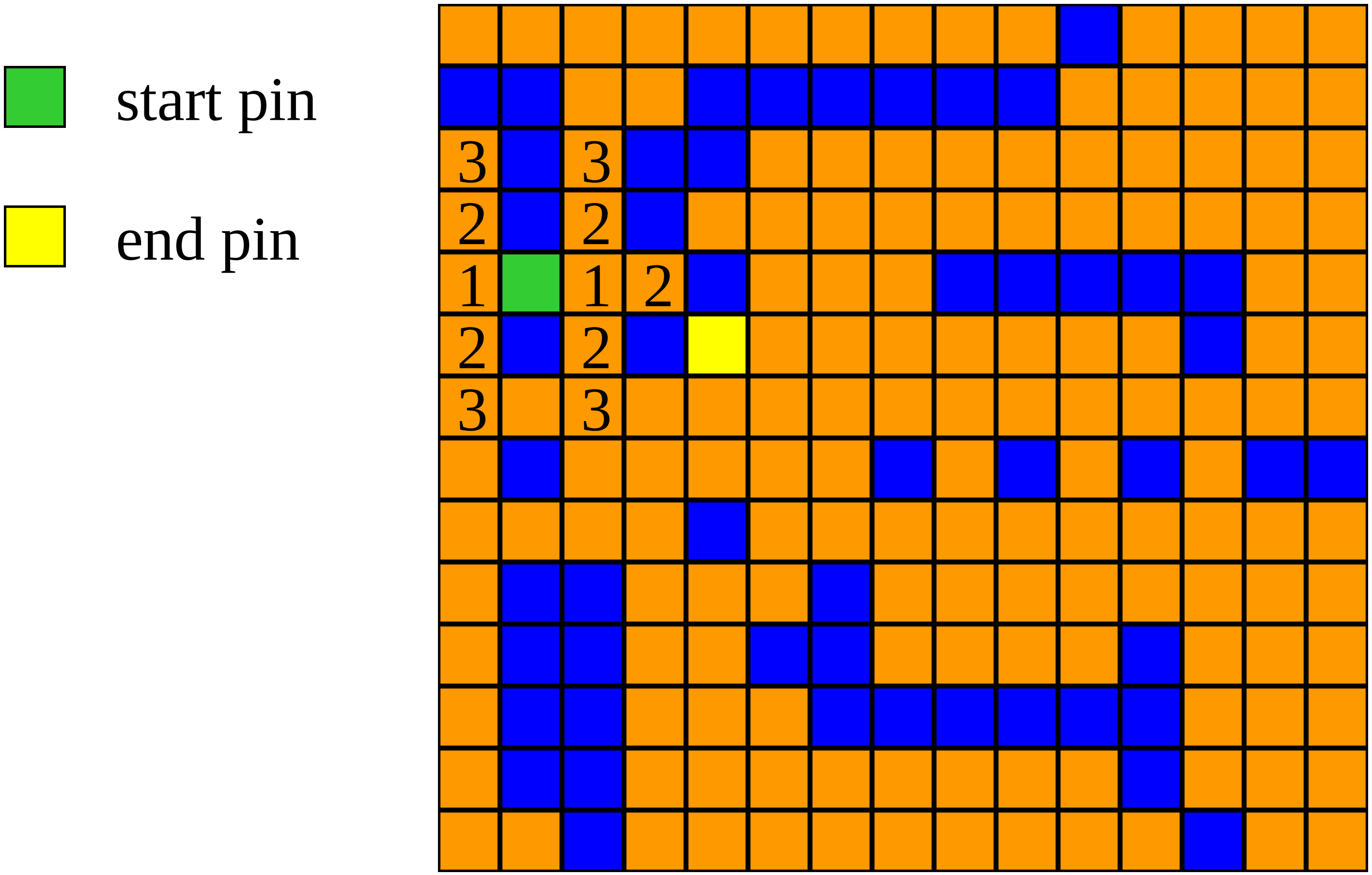
Label all reachable unlabeled squares 2 units from start.

Lee's Wire Router



Label all reachable unlabeled squares 3 units from start.

Lee's Wire Router

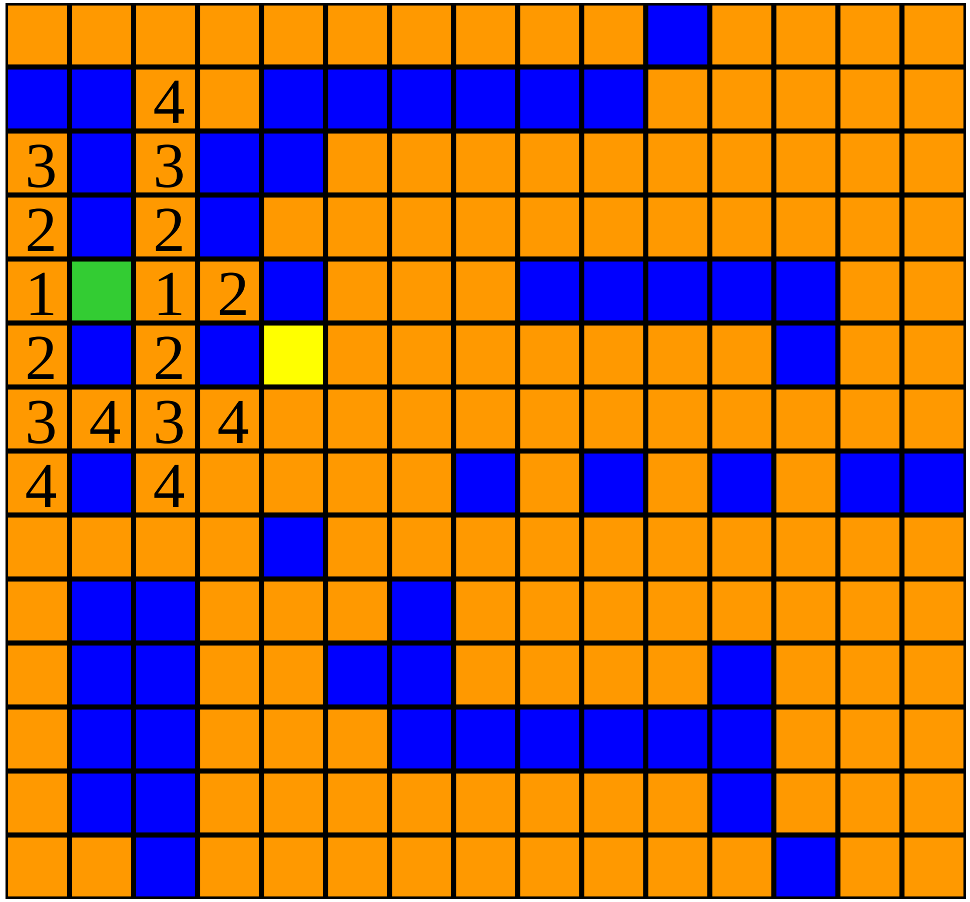


Label all reachable unlabeled squares 4 units from start.

Lee’s Wire Router

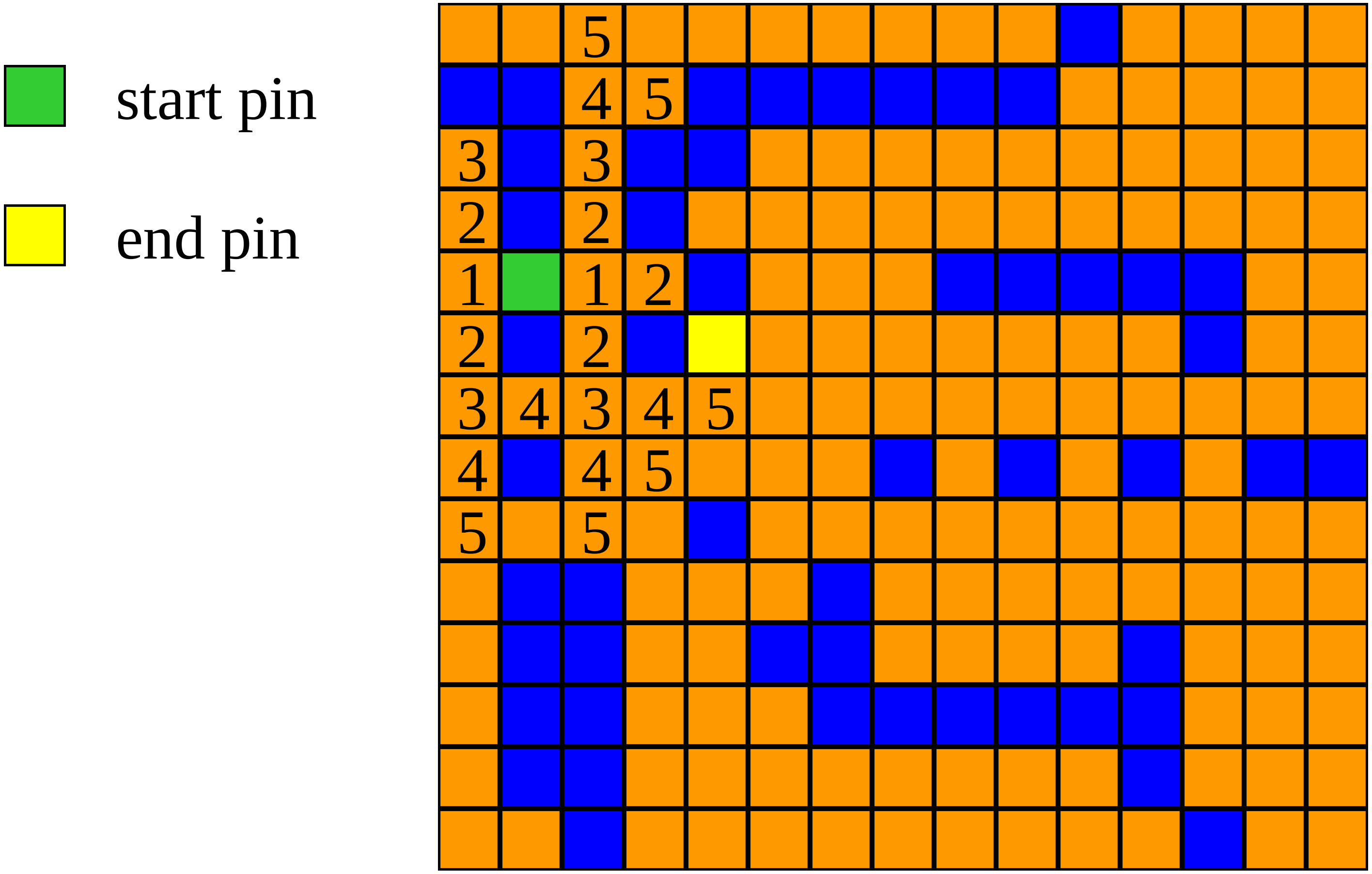
start pin

end pin



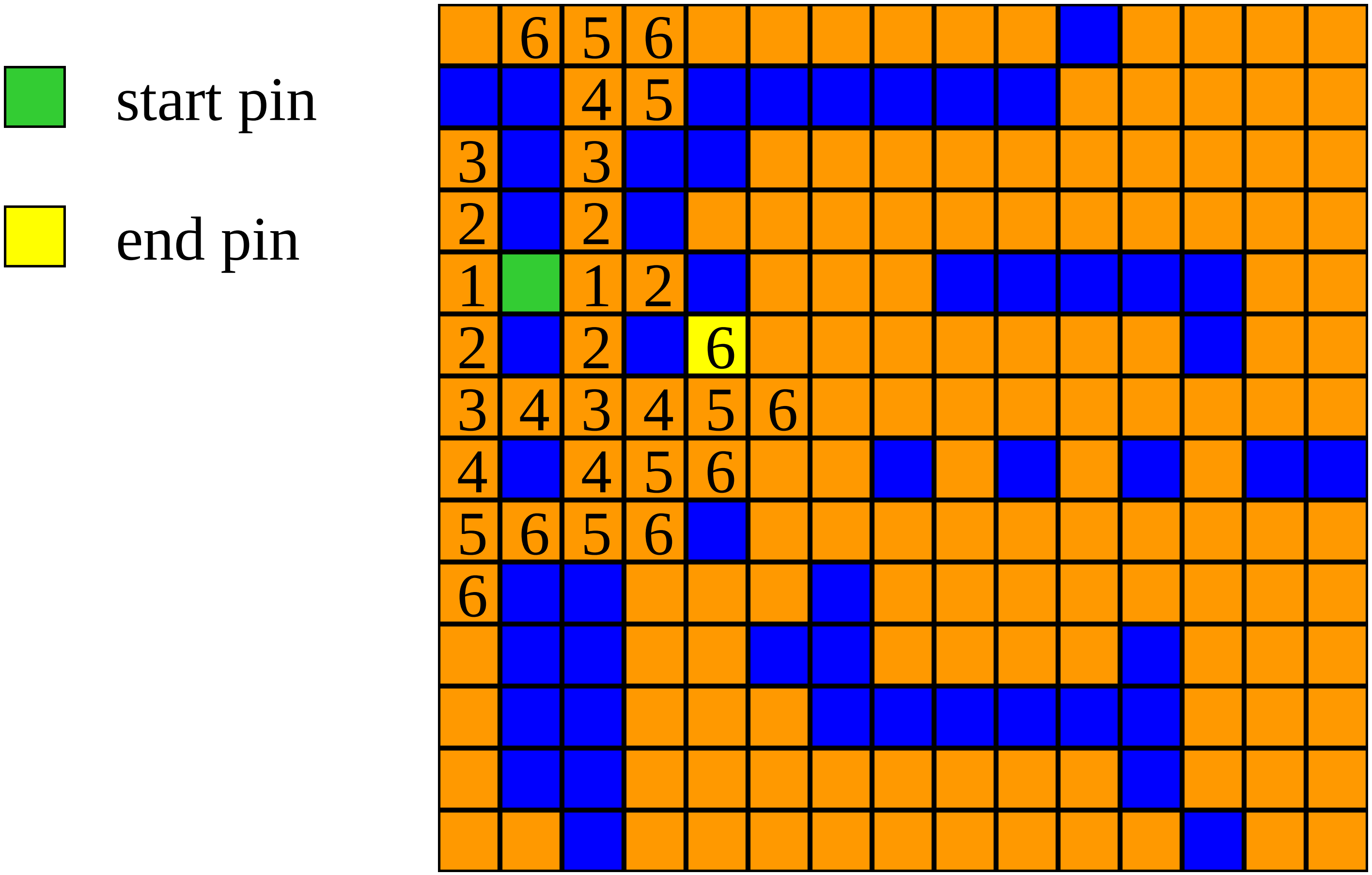
Label all reachable unlabeled squares 5 units from start.

Lee's Wire Router



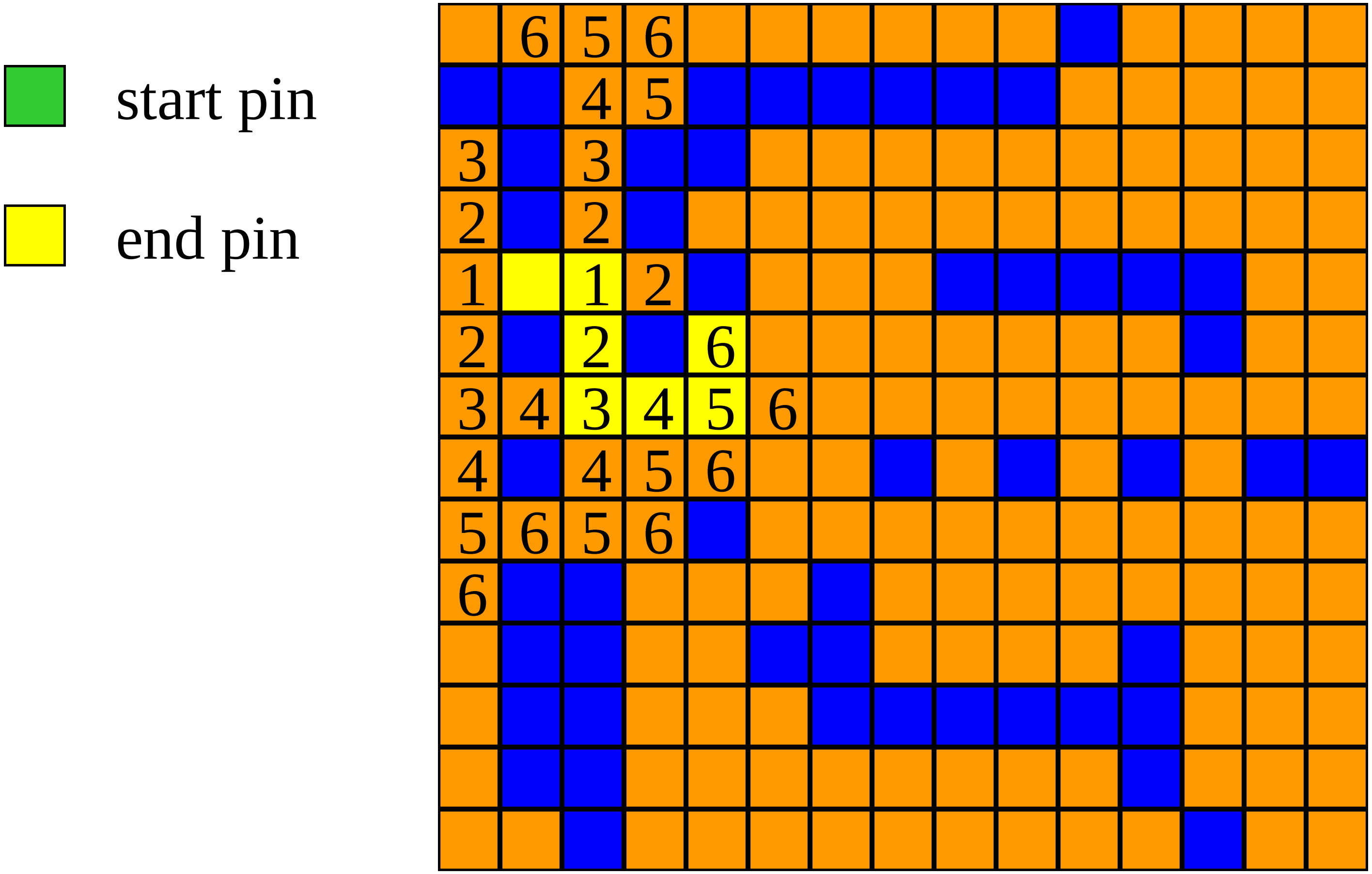
Label all reachable unlabeled squares 6 units from start.

Lee's Wire Router



End pin reached. Traceback.

Lee's Wire Router



End pin reached. Traceback.

Queue Operations

- IsFullQ … return true iff queue is full
- IsEmptyQ … return true iff queue is empty
- AddQ … add an element at the rear of the queue
- DeleteQ … delete and return the front element of the queue

Queue in an Array

– Use a 1D array to represent a queue.
– Suppose queue elements are stored with the front element in queue[0], the next in queue[1], and so on.

Queue in an Array

a	b	c	d	e										

0 1 2 3 4 5 6

- DeleteQ() => delete queue[0]
  - O(queue size) time
- AddQ(x) => if there is capacity, add at right end
  - O(1) time

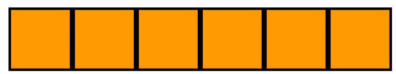
O(1) AddQ and DeleteQ

– to perform each opertion in O(1) time (excluding array doubling), we use a circular representation.

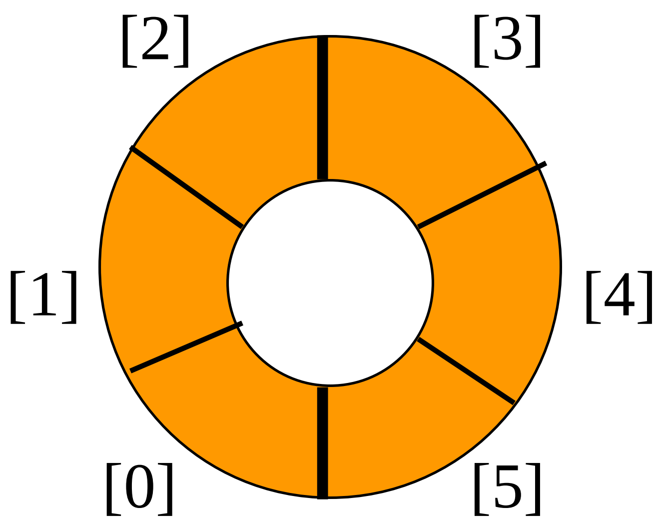
Circular Array

- Use a 1D array queue.

queue[]

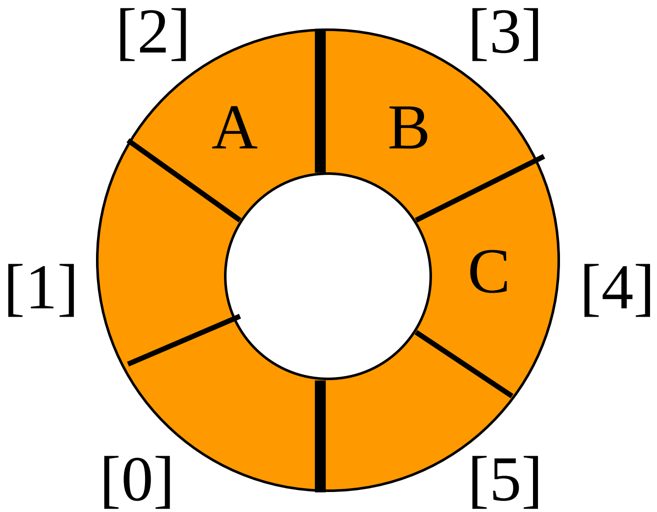


- Circular view of array.



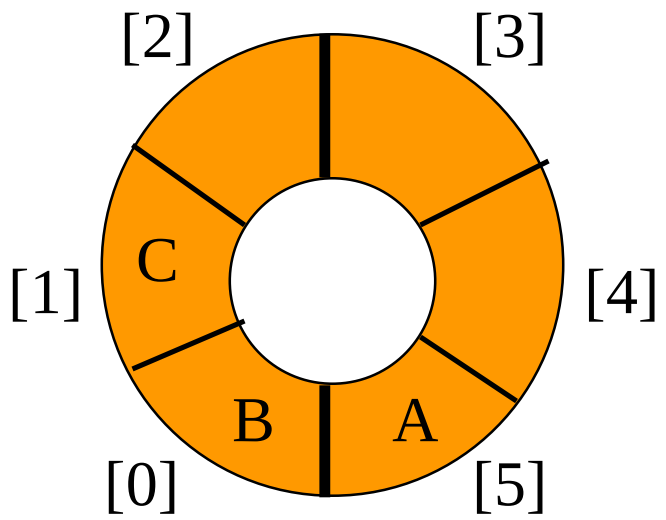
Circular Array

- Possible configuration with 3 elements.



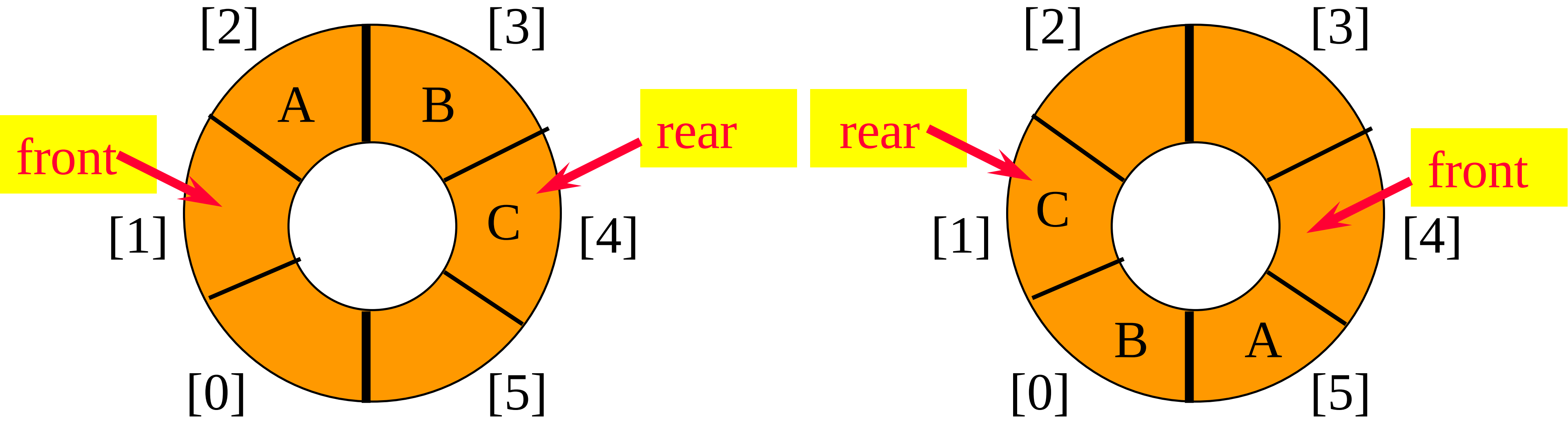
Circular Array

- Another possible configuration with 3 elements.



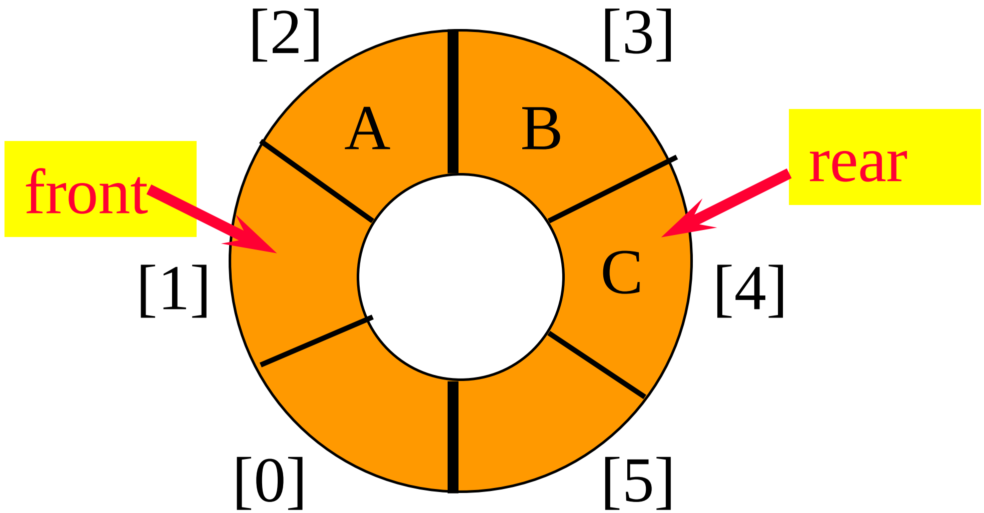
Circular Array

- Use integer variables front and rear.
  - front is one position counterclockwise from first element
  - rear gives position of last element



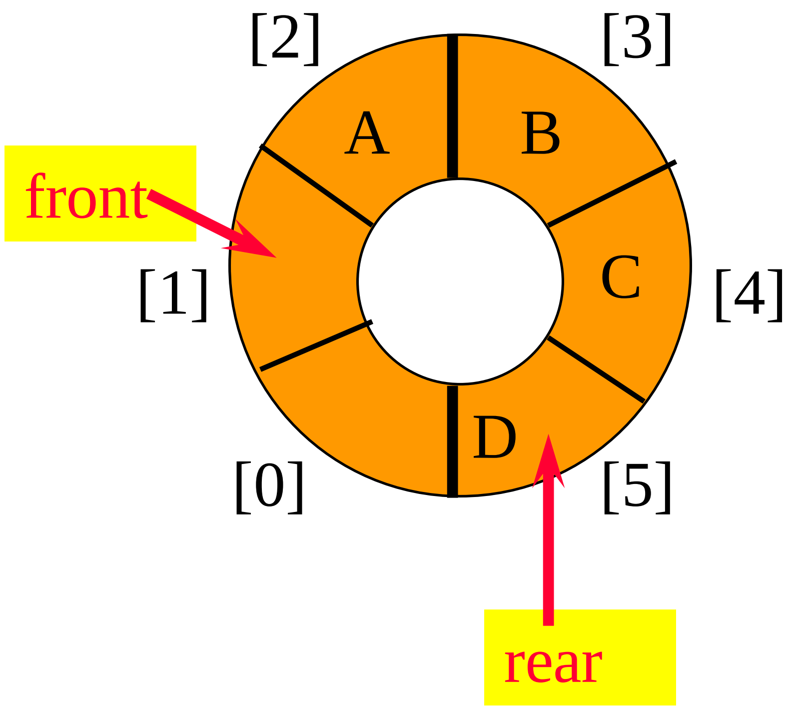
Add An Element

- Move rear one clockwise.



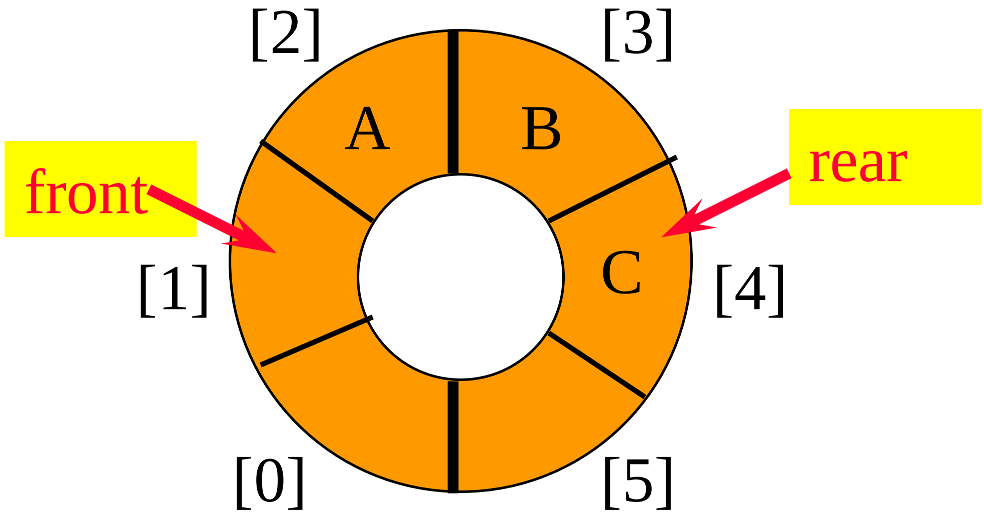
Add An Element

- Move rear one clockwise.
- Then put into queue[rear].



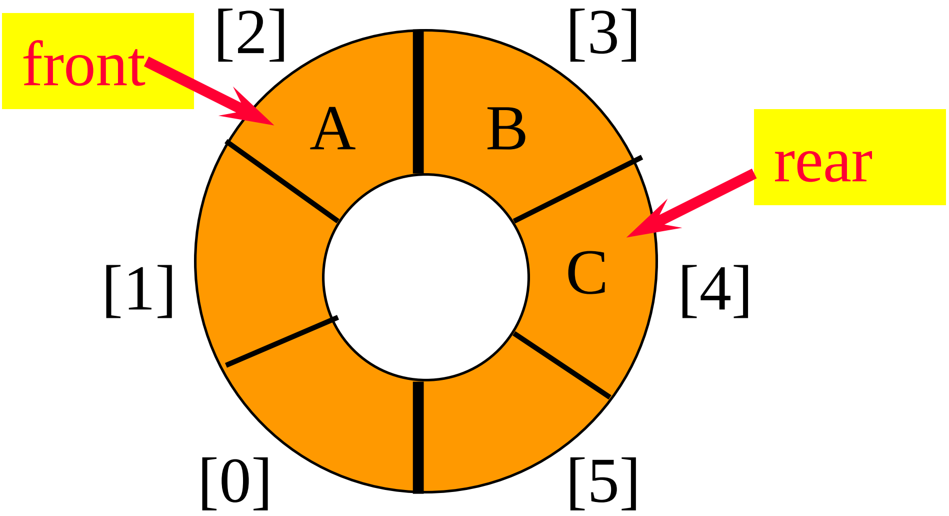
Delete An Element

- Move front one clockwise.



Delete An Element

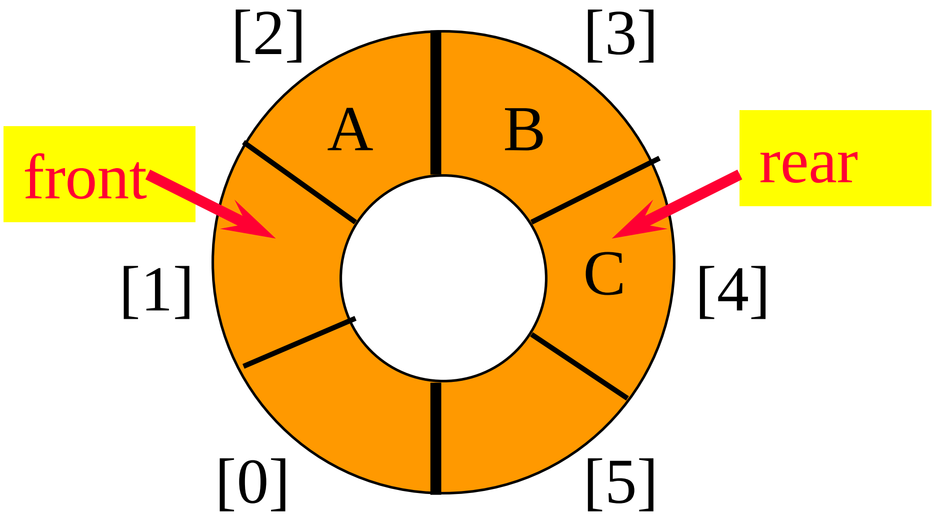
- Move front one clockwise.
- Then extract from queue[front].



Moving rear Clockwise

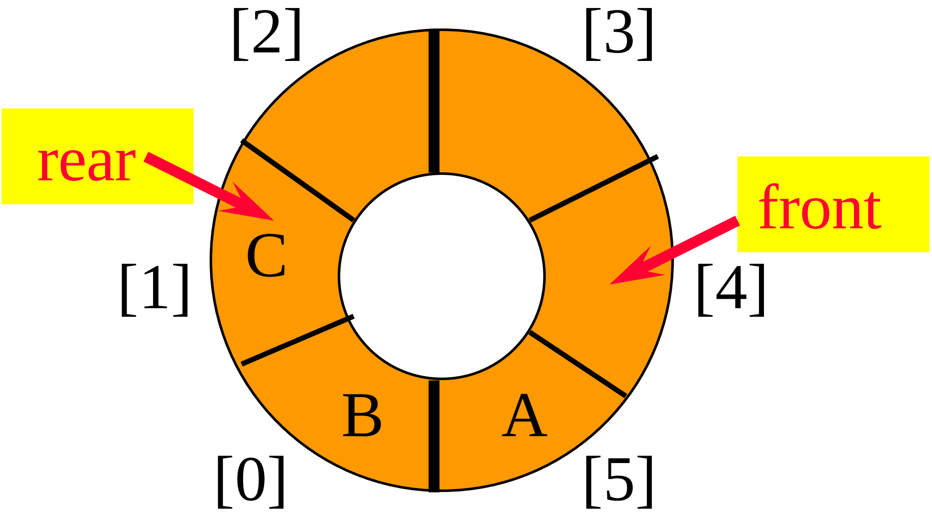
- rear++;

  if (rear = = capacity) rear = 0;

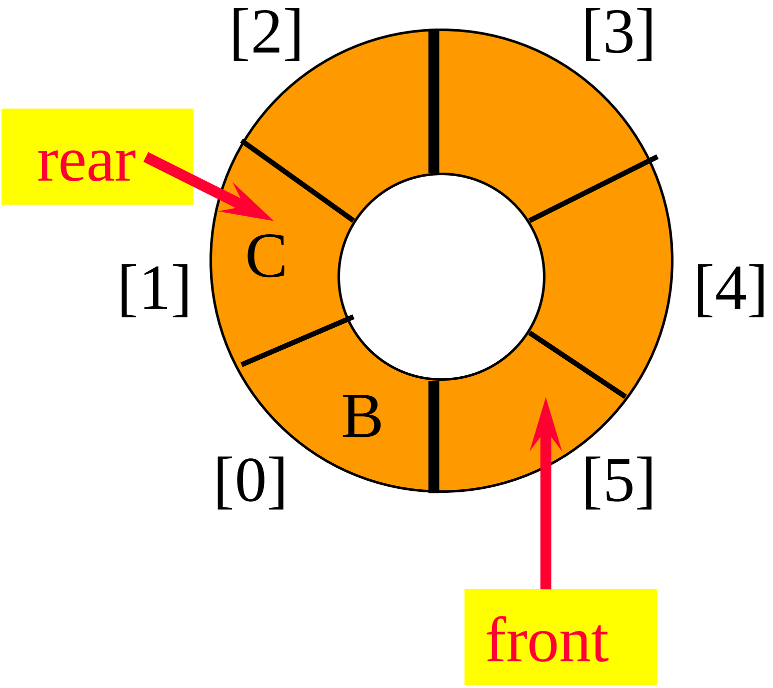


- rear = (rear + 1) % capacity;

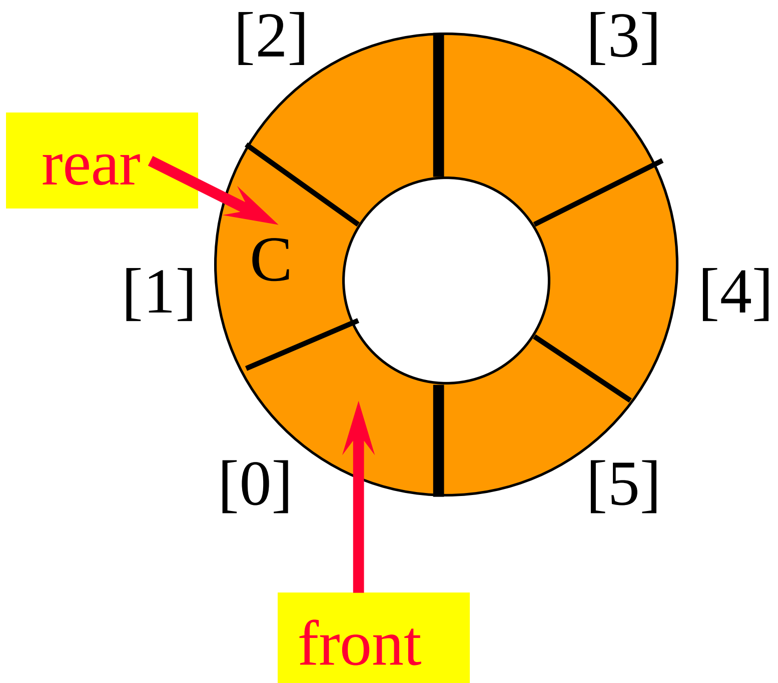
Empty That Queue



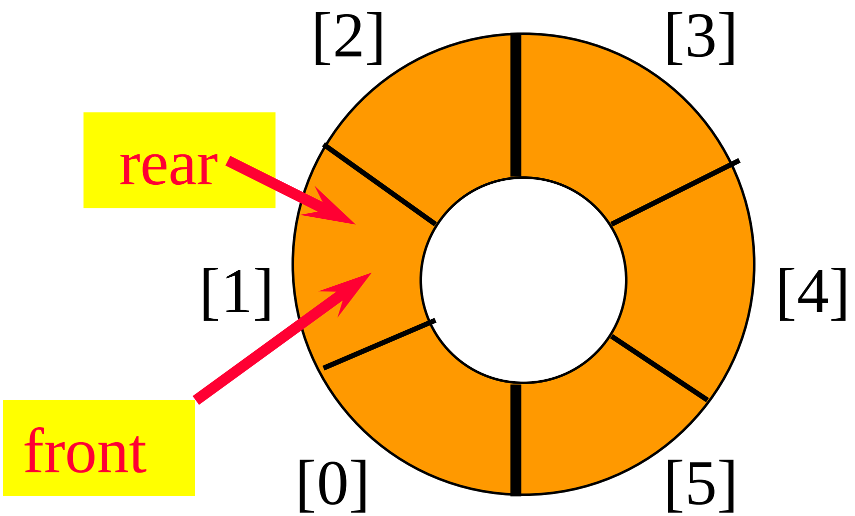
Empty That Queue



Empty That Queue

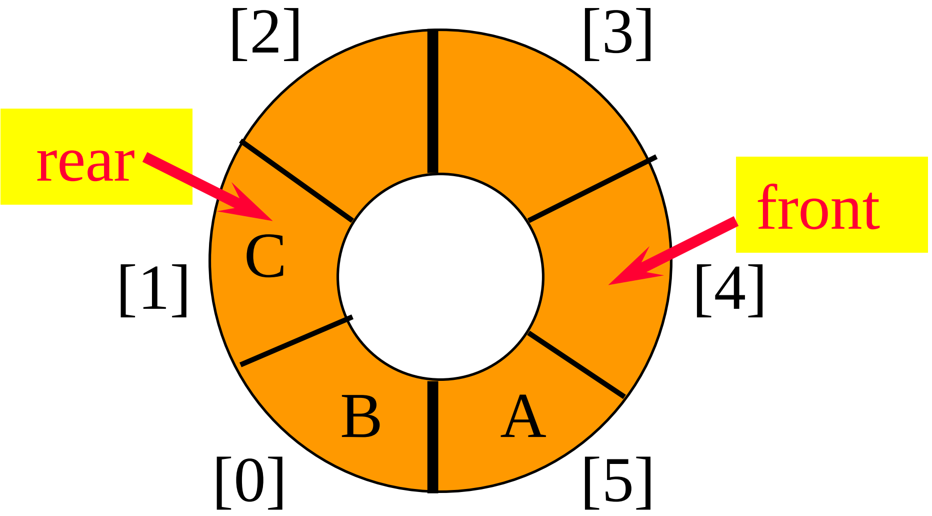


Empty That Queue

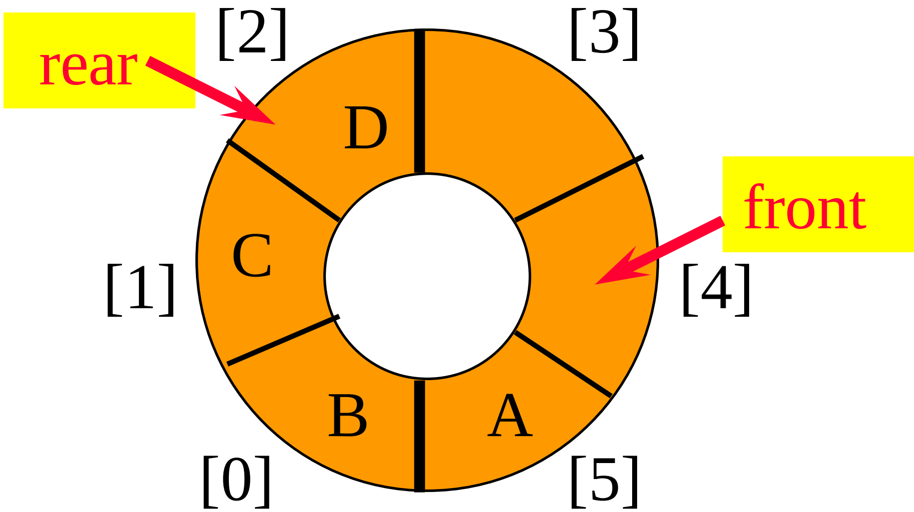


- When a series of removes causes the queue to become empty, front = rear.
- When a queue is constructed, it is empty.
- So initialize front = rear = 0.

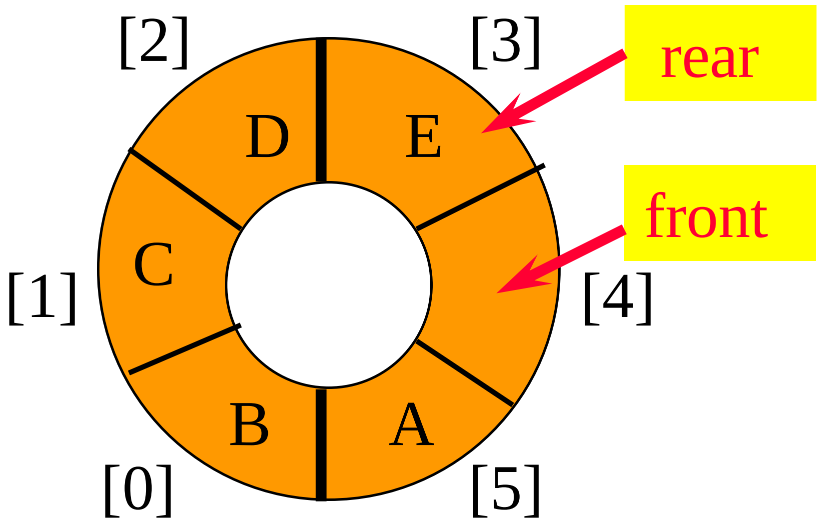
A Full Tank Please



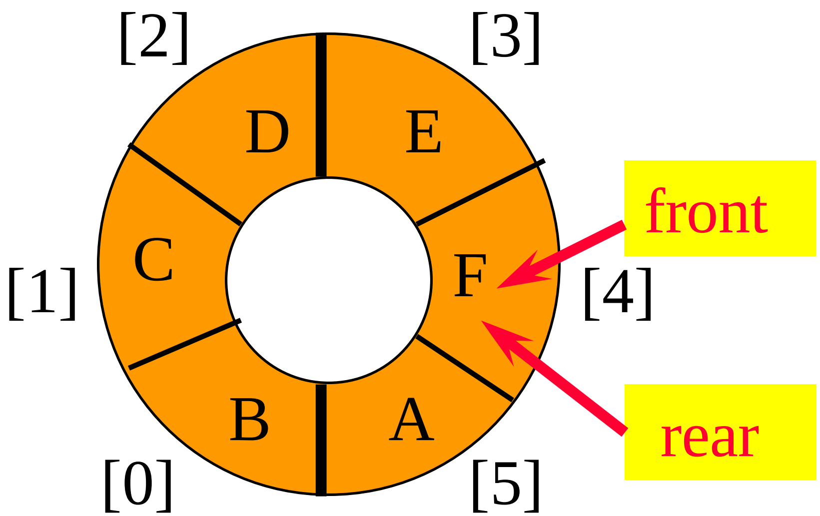
A Full Tank Please



A Full Tank Please



A Full Tank Please



- When a series of adds causes the queue to become full, front = rear.
- So we cannot distinguish between a full queue and an empty queue!

Ouch!!!!!

- Remedies.
  - Don’t let the queue get full.
    - When the addition of an element will cause the queue to be full, increase array size.
    - This is what the text does.
  - Define a boolean variable lastOperationIsAddQ.
    - Following each AddQ set this variable to true.
    - Following each DeleteQ set to false.
    - Queue is empty iff (front == rear) && !lastOperationIsAddQ
    - Queue is full iff (front == rear) && lastOperationIsAddQ

Ouch!!!!!

- Remedies (continued).
  - Define an integer variable size.
    - Following each AddQ do size++.
    - Following each DeleteQ do size--.
    - Queue is empty iff (size == 0)
    - Queue is full iff (size == arrayLength)
  - Performance is slightly better when first strategy is used.

Priority Queues

Two kinds of priority queues:

- Min priority queue.
- Max priority queue.

Min Priority Queue

- Collection of elements.
- Each element has a priority or key.
- Supports following operations:
  - empty
  - size
  - insert an element into the priority queue (push)
  - get element with min priority (top)
  - remove element with min priority (pop)

Max Priority Queue

- Collection of elements.
- Each element has a priority or key.
- Supports following operations:
  - empty
  - size
  - insert an element into the priority queue (push)
  - get element with max priority (top)
  - remove element with max priority (pop)

Complexity Of Operations

Use a heap or a leftist tree (both are defined later).

empty, size, and top => O(1) time

insert (push) and remove (pop) => O(log n) time where n is the size of the priority queue

Applications

Sorting

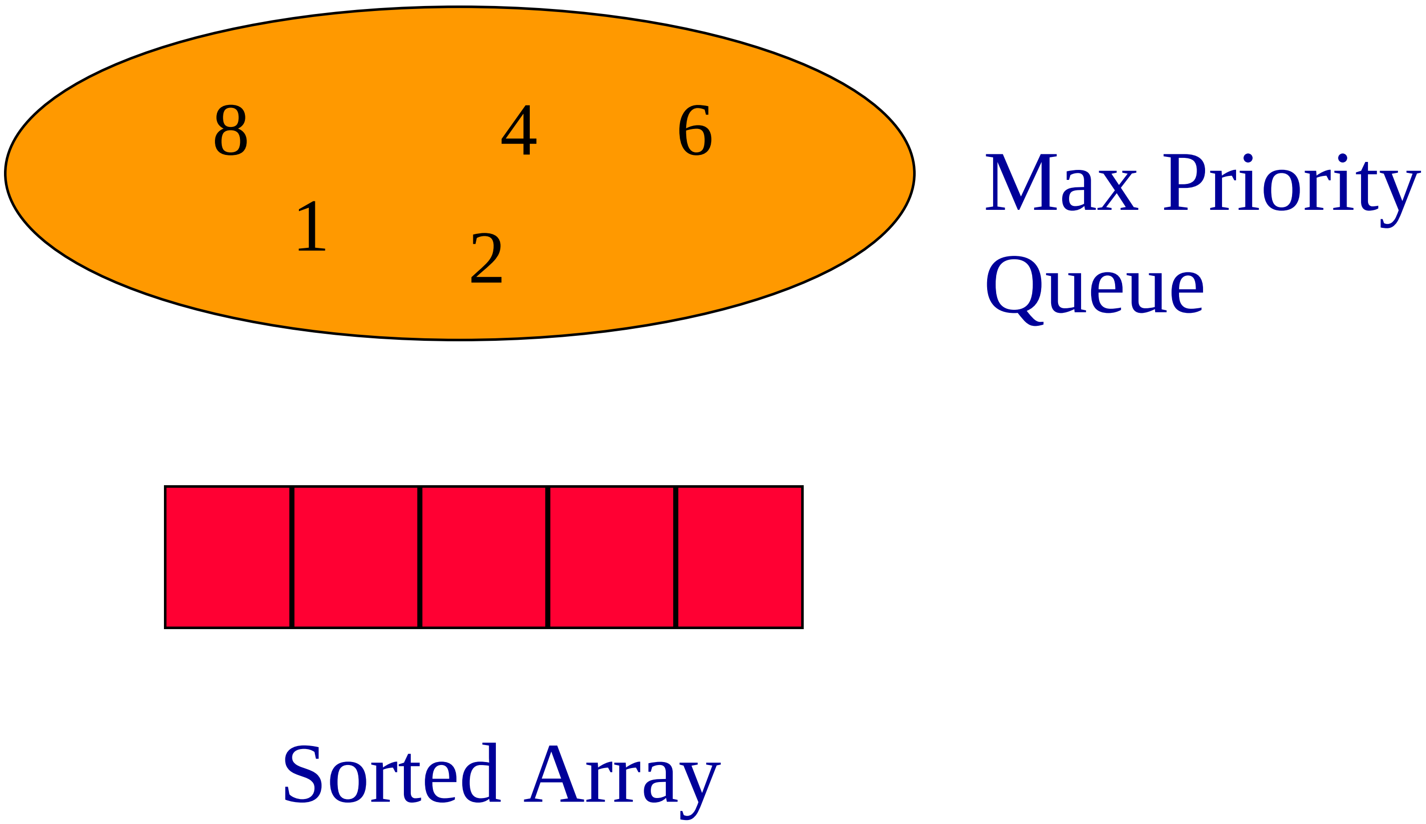
- use element key as priority
- insert elements to be sorted into a priority queue
- remove/pop elements in priority order
  - if a min priority queue is used, elements are extracted in ascending order of priority (or key)
  - if a max priority queue is used, elements are extracted in descending order of priority (or key)

Sorting Example

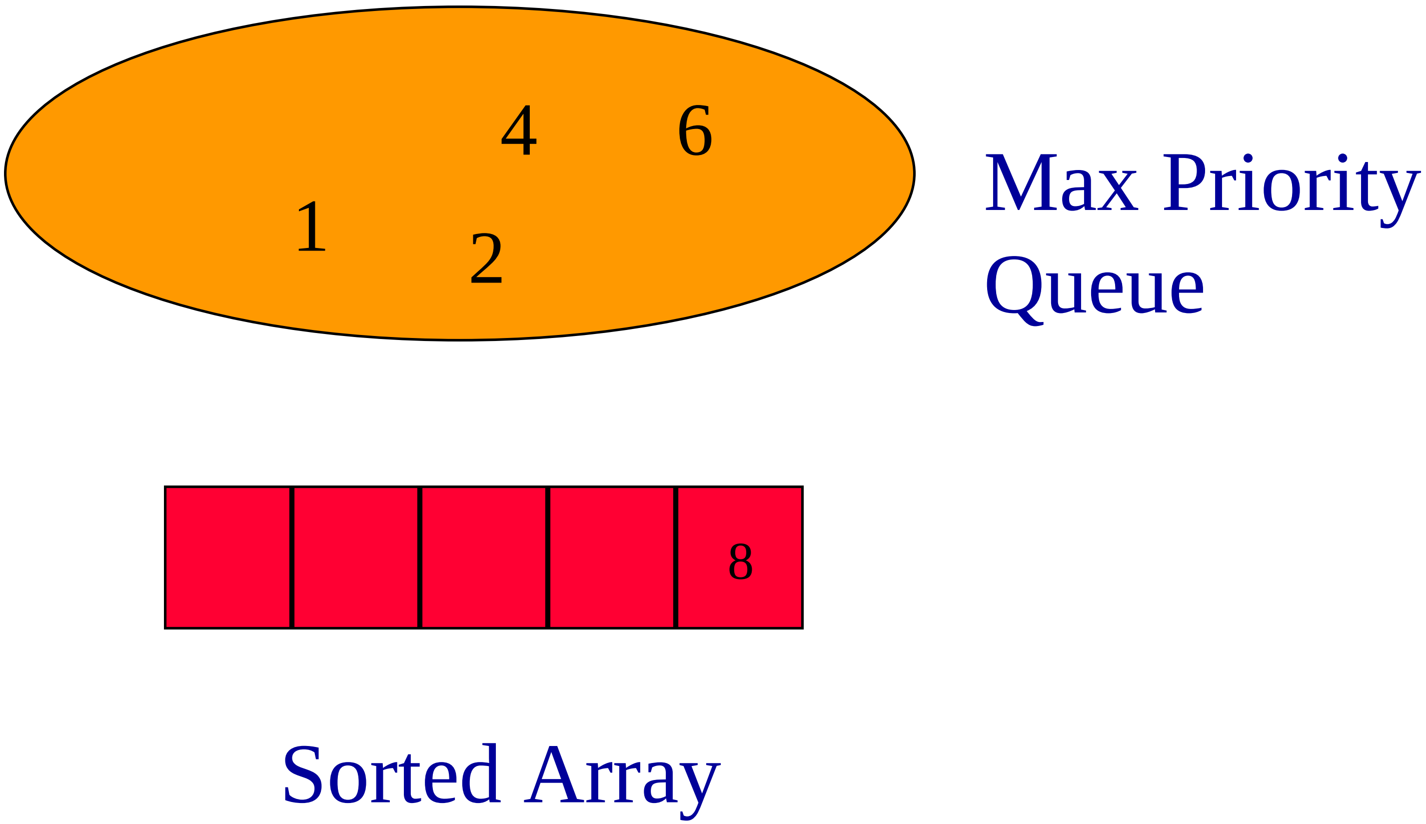
Sort five elements whose keys are 6, 8, 2, 4, 1 using a max priority queue.

- Insert the five elements into a max priority queue.
- Do five remove max operations placing removed elements into the sorted array from right to left.

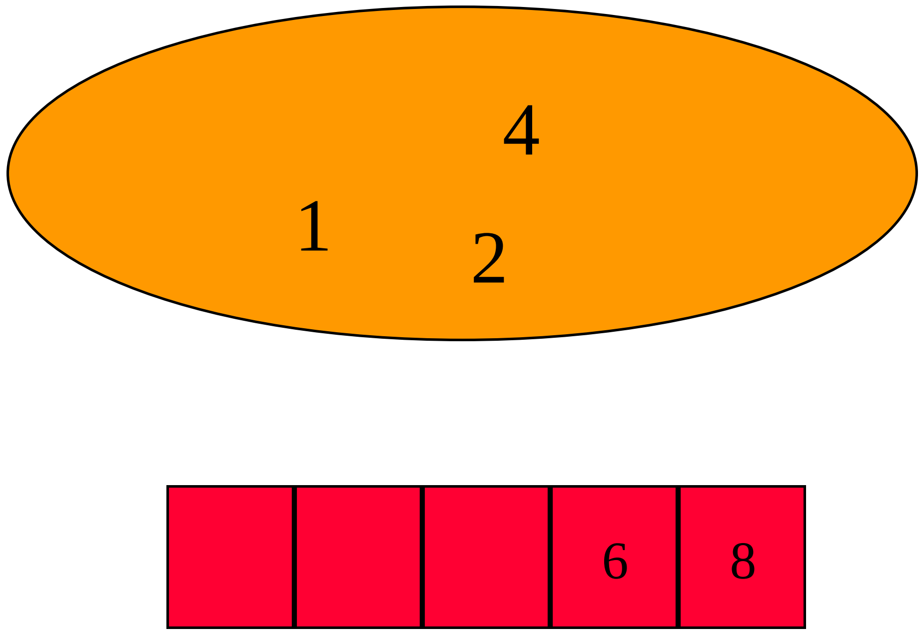
After Inserting Into Max Priority Queue



After First Remove Max Operation



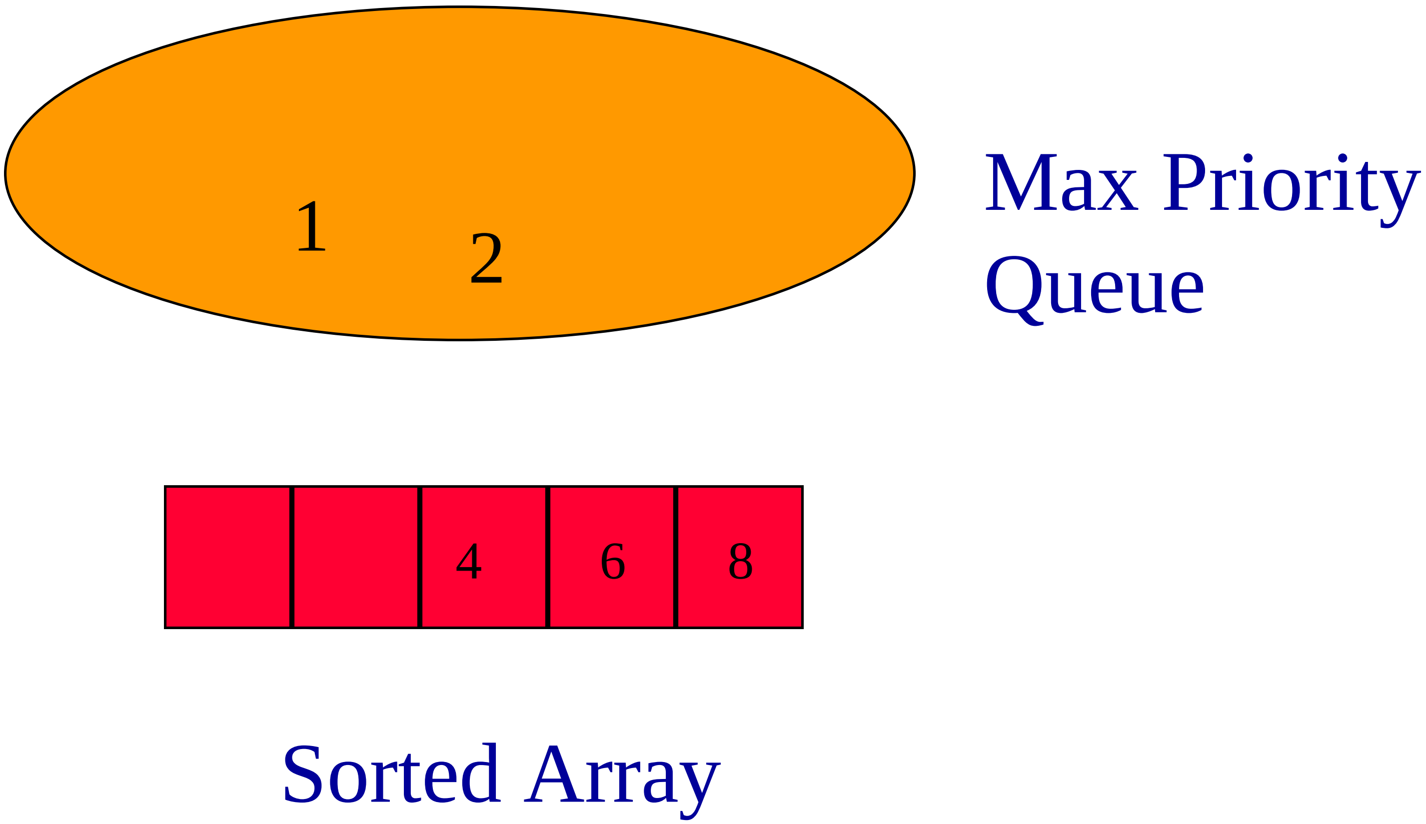
After Second Remove Max Operation



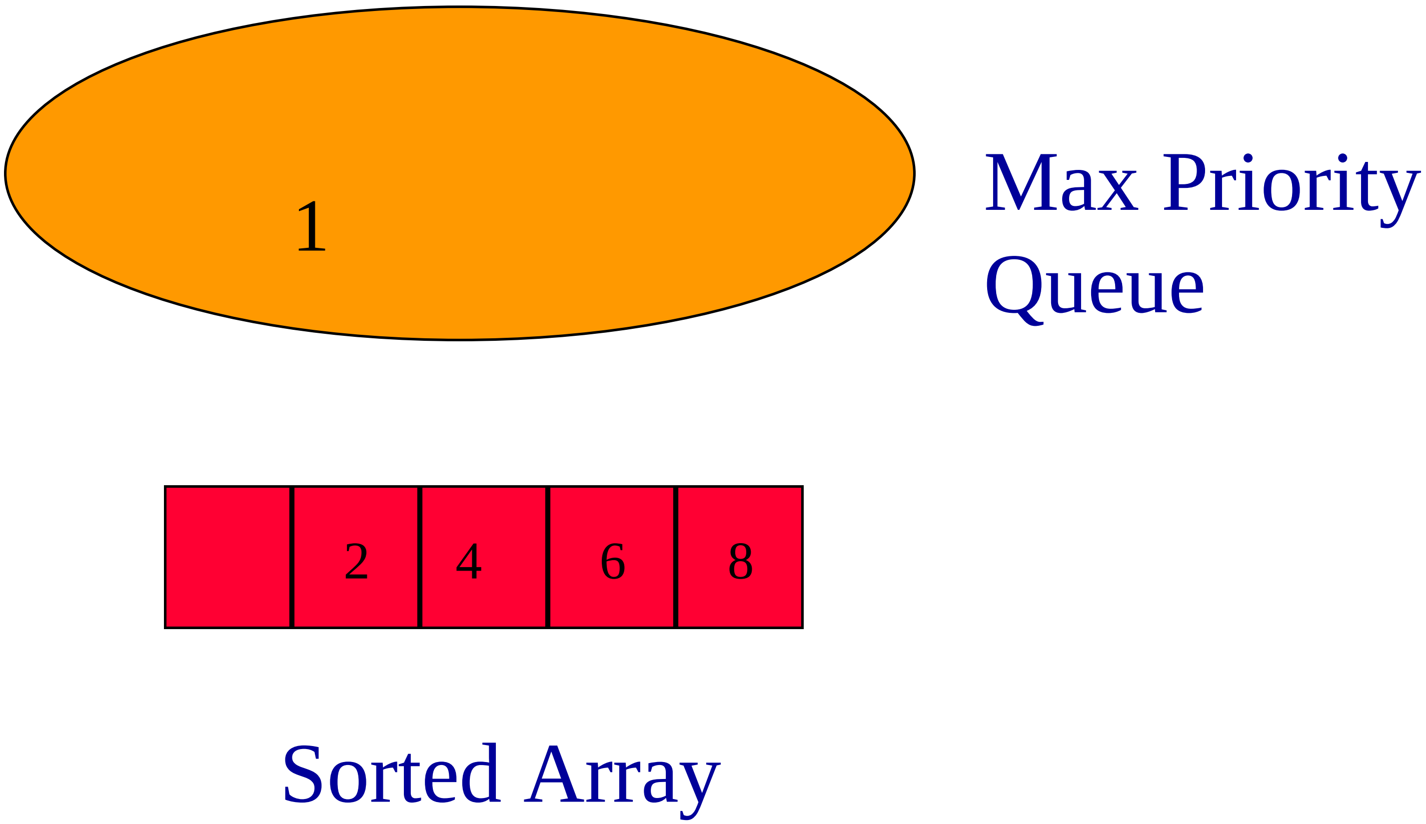
Max Priority Queue

Sorted Array

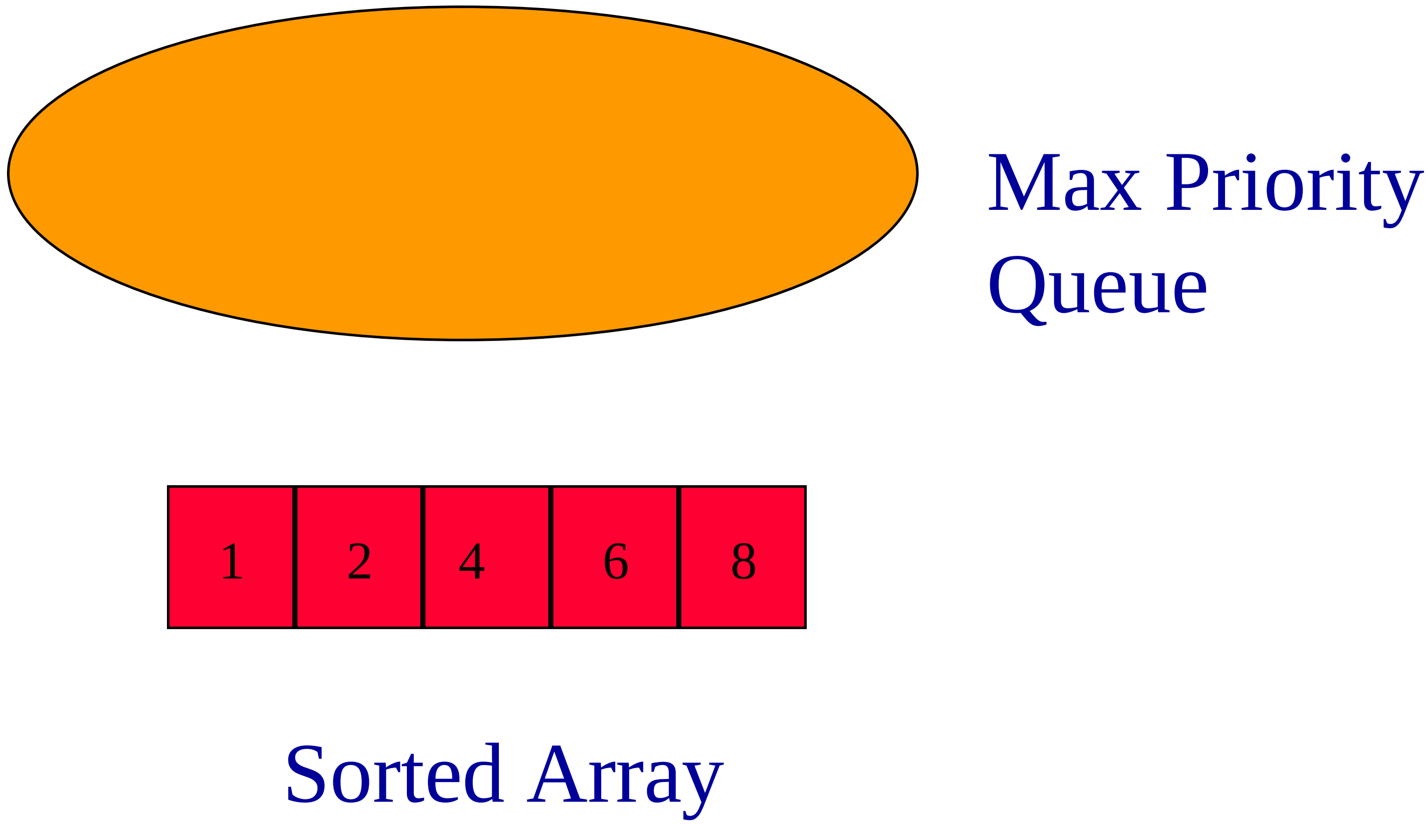
After Third Remove Max Operation



After Fourth Remove Max Operation



After Fifth Remove Max Operation



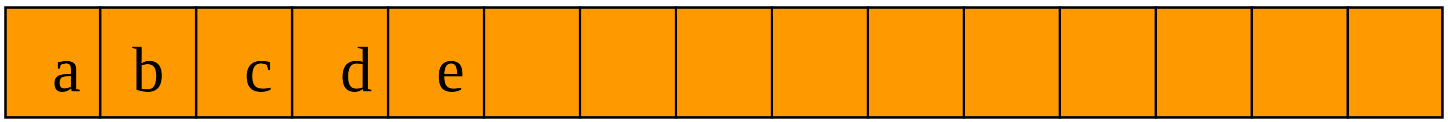
Linked Lists

- list elements are stored, in memory, in an arbitrary order

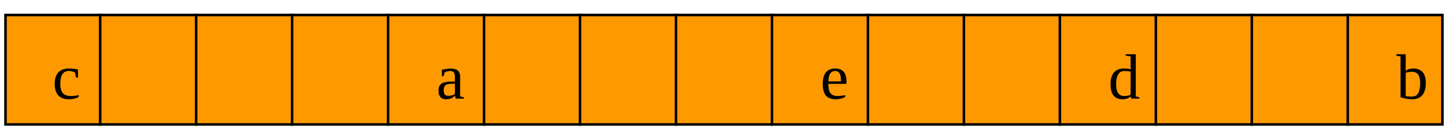
- explicit information (called a link) is used to go from one element to the next

Memory Layout

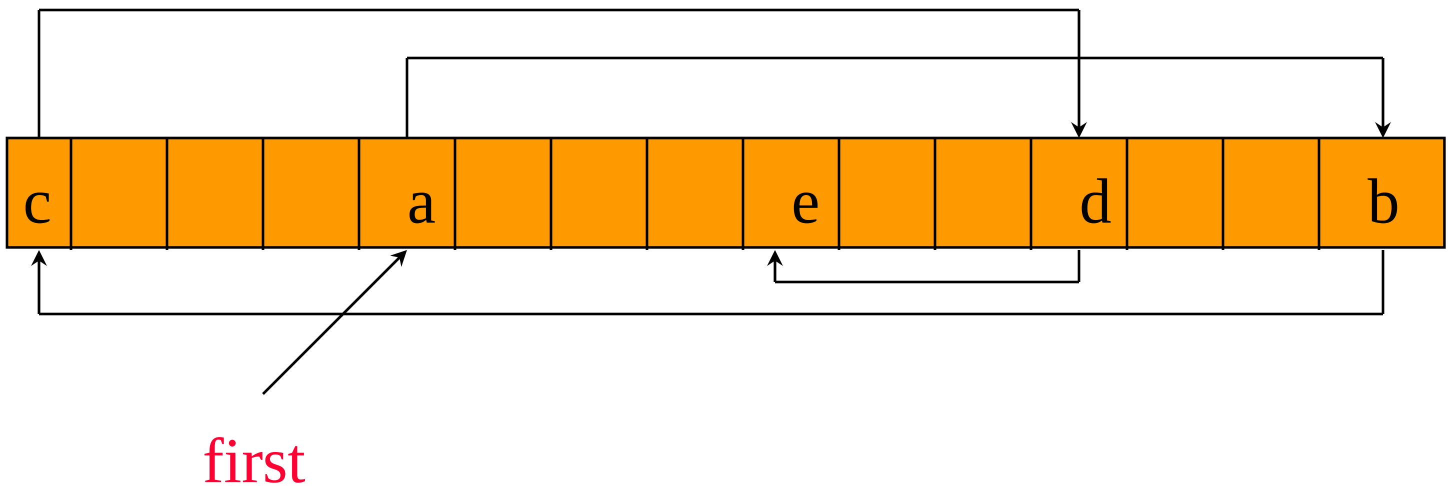
Layout of L = (a,b,c,d,e) using an array representation.



A linked representation uses an arbitrary layout.



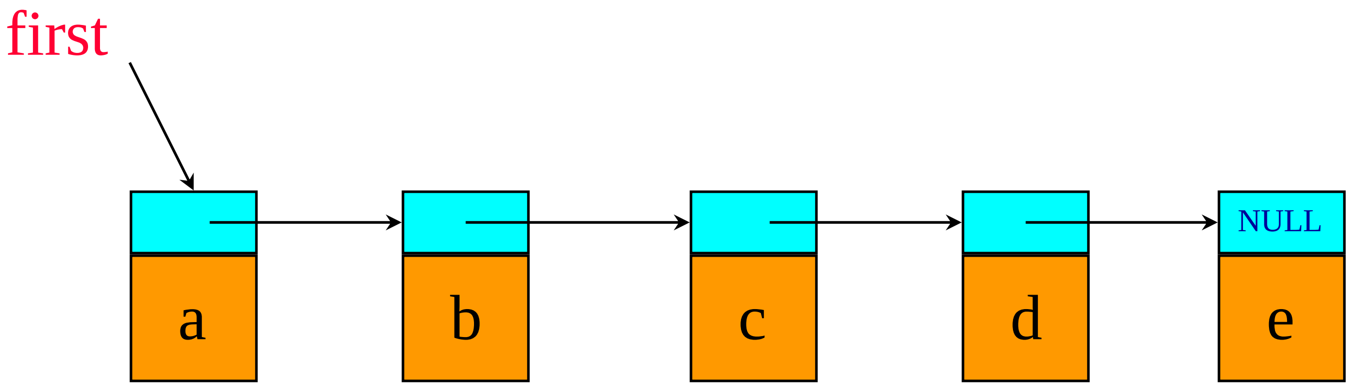
Linked Representation

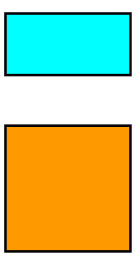


pointer (or link) in e is NULL

use a variable first to get to the first element a

Normal Way To Draw A Linked List

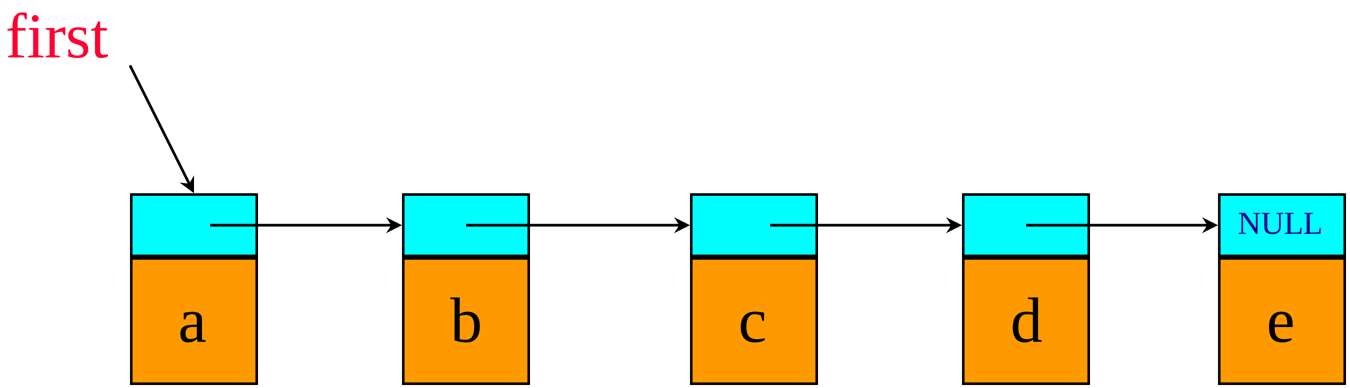




link or pointer field of node

data field of node

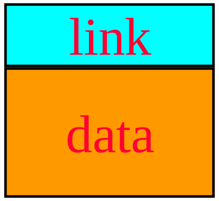
Chain



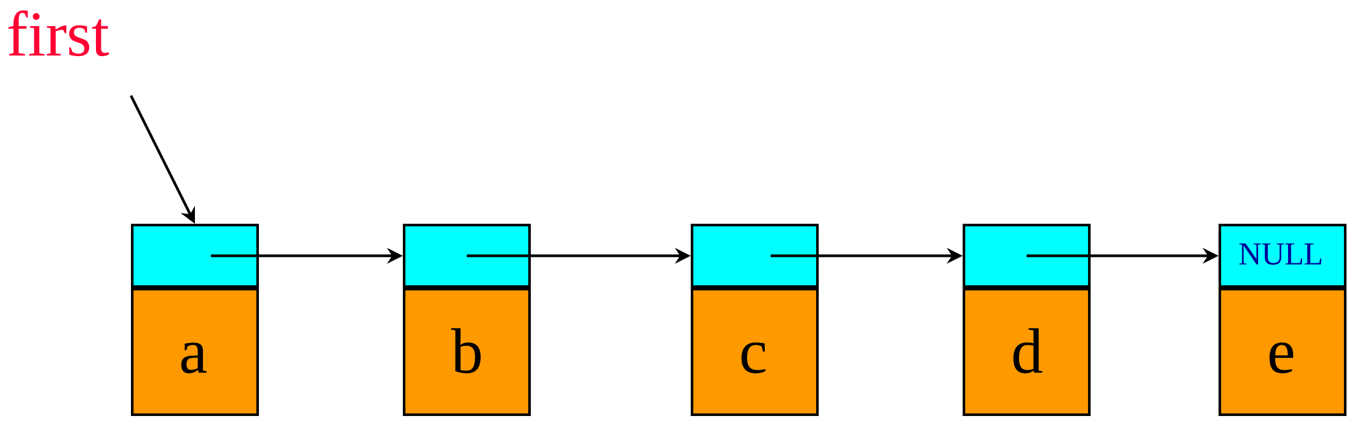
- A chain is a linked list in which each node represents one element.
- There is a link or pointer from one element to the next.
- The last node has a NULL (or 0) pointer.

Node Representation

```
typedef struct listNode *listPointer;
typedef struct {
        char data;
        listPointer link;
        } listNode;
```



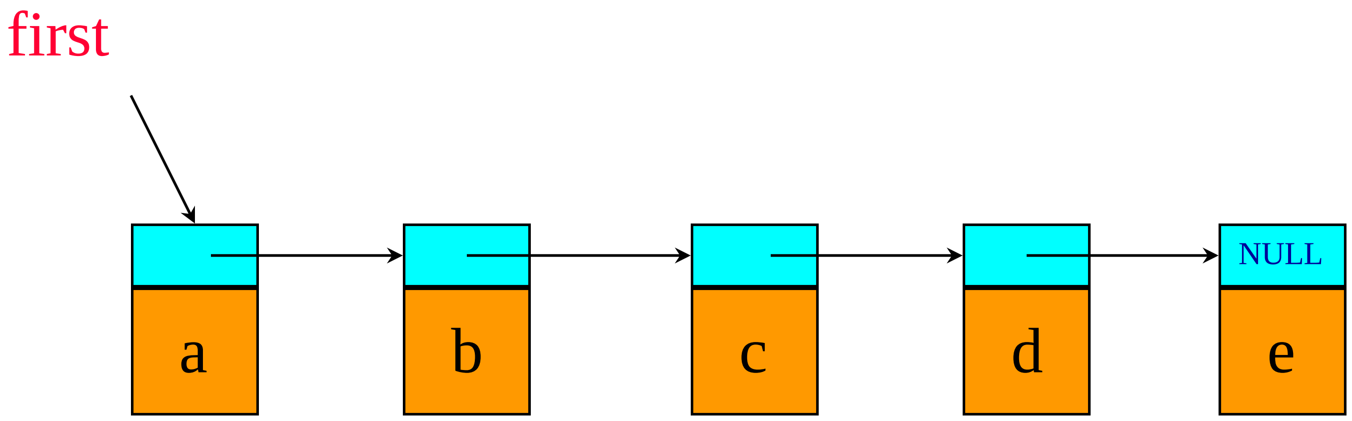
get(0)



desiredNode = first; // gets you to first node

return desiredNode–>data;

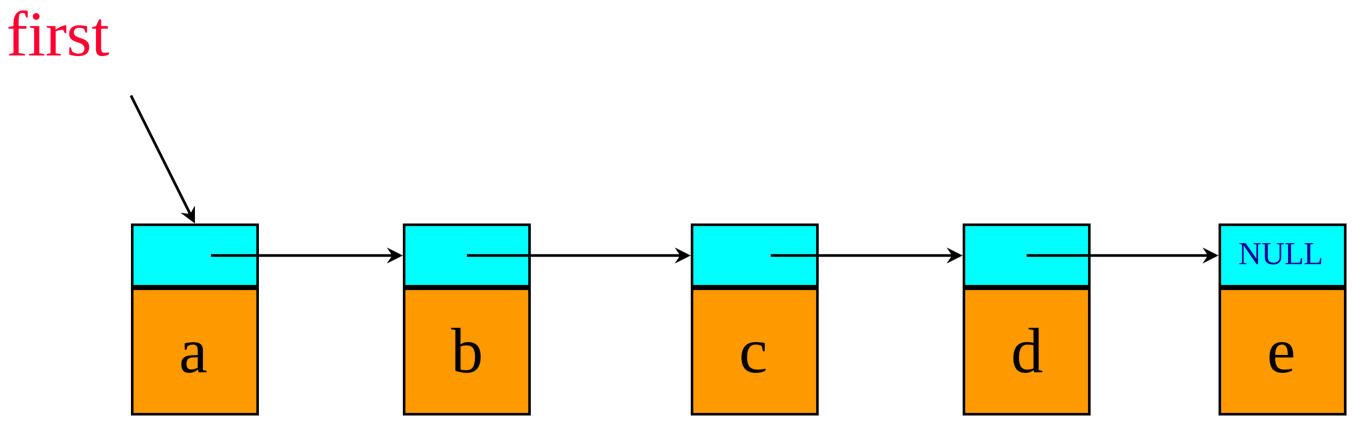
get(1)



desiredNode = first–>link; // gets you to second node

return desiredNode–>data;

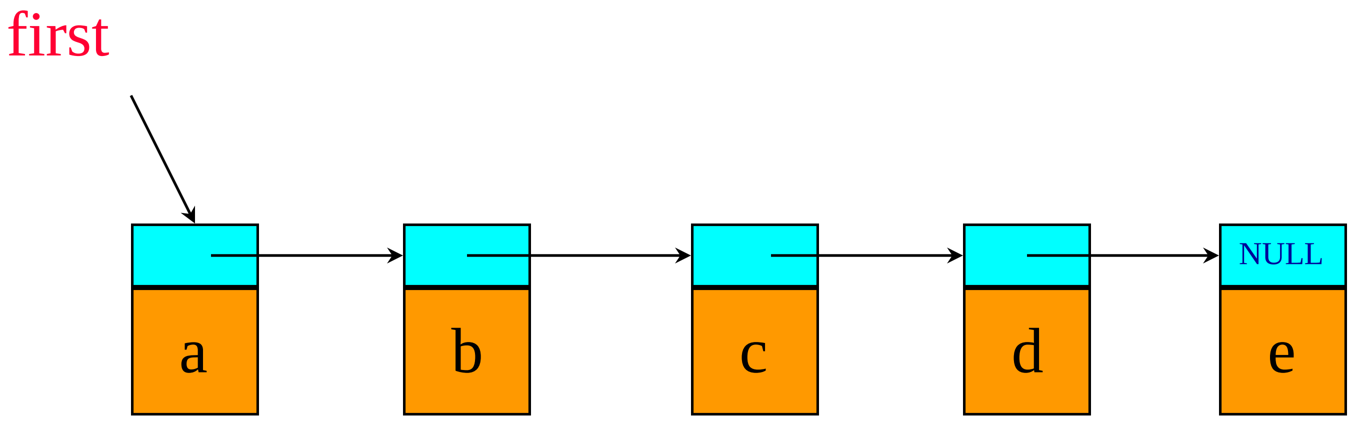
get(2)



desiredNode = first–>link–>link; // gets you to third node

return desiredNode–>data;

get(5)

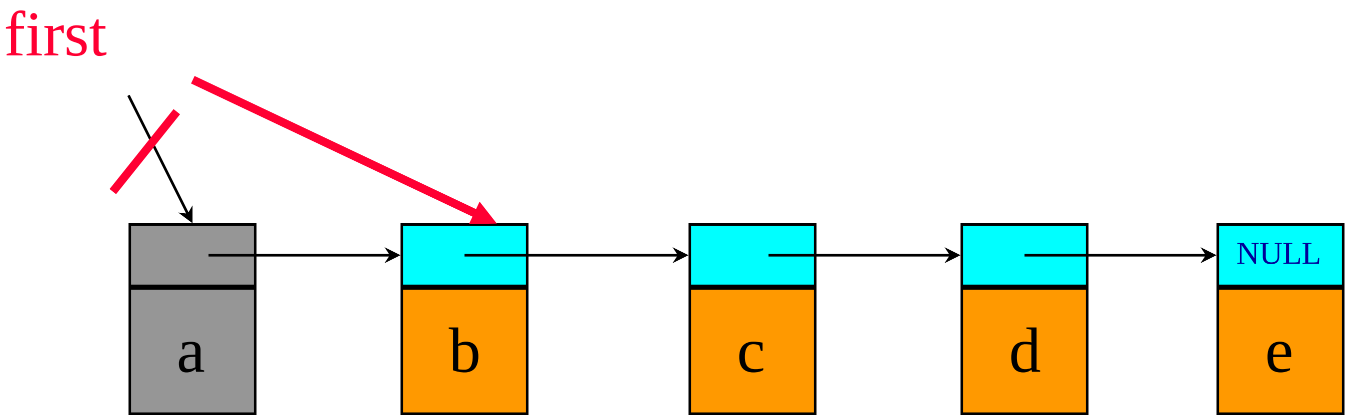


desiredNode = first–>link–>link–>link–>link–>link;

// desiredNode = NULL

return desiredNode–>data;   // NULL.element

Delete An Element



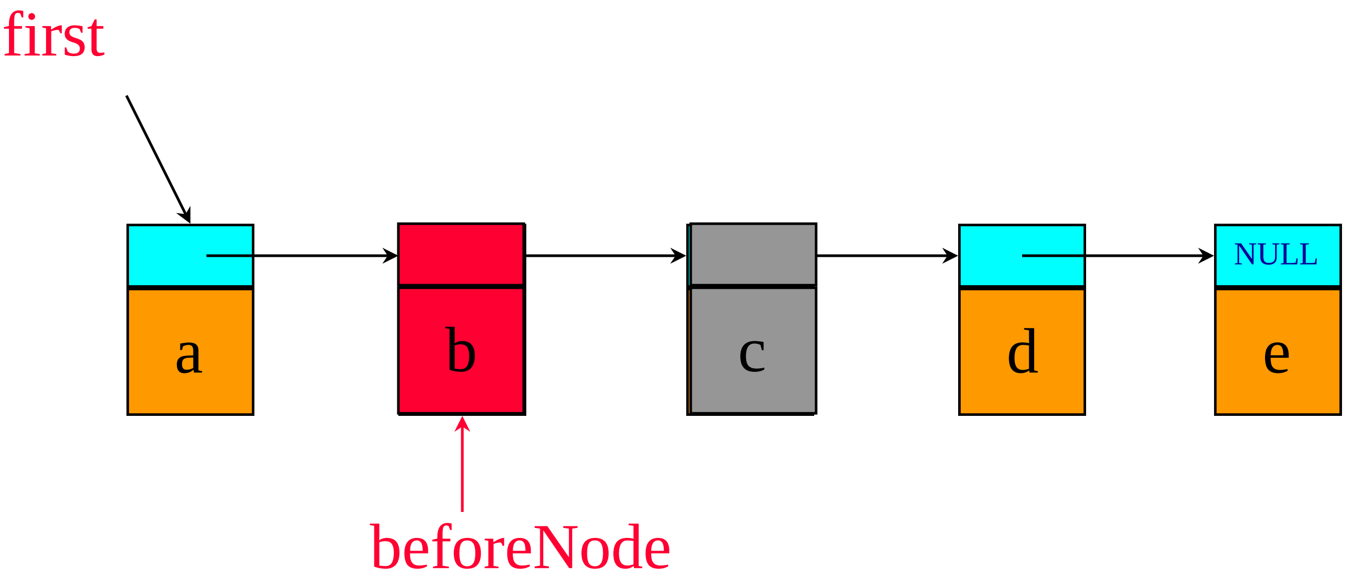
delete(0)

deleteNode = first;

first = first–>link;

free(deleteNode);

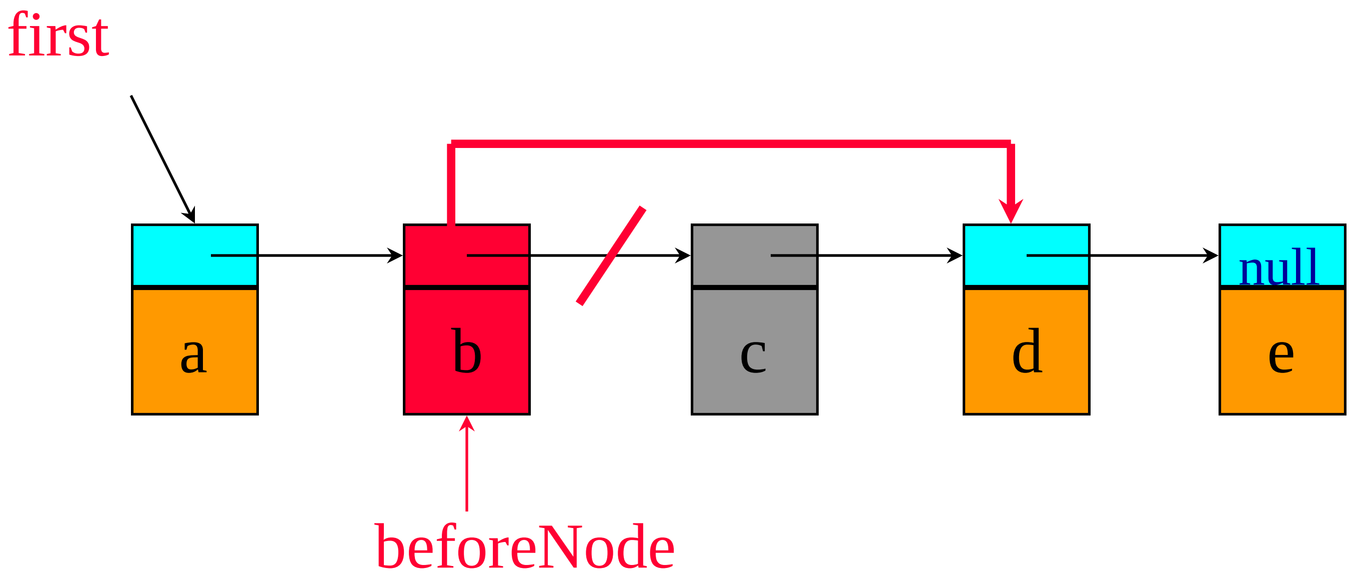
delete(2)



first get to node just before node to be removed

beforeNode = first–>link;

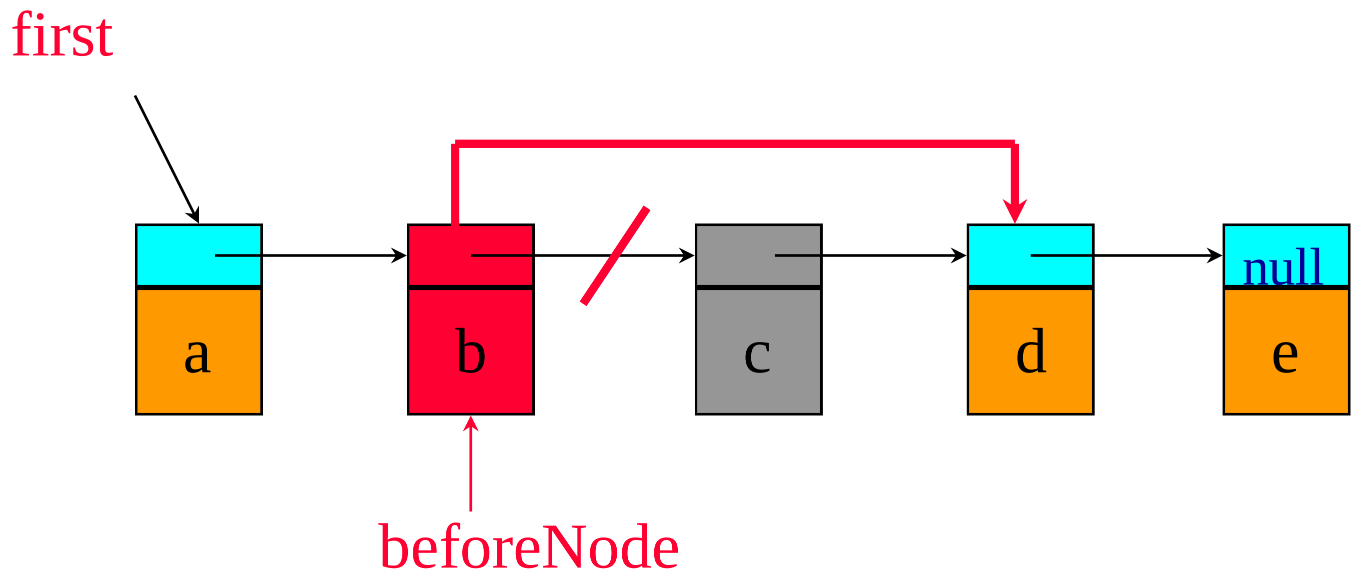
delete(2)



save pointer to node that will be deleted

deleteNode = beforeNode–>link;

delete(2)

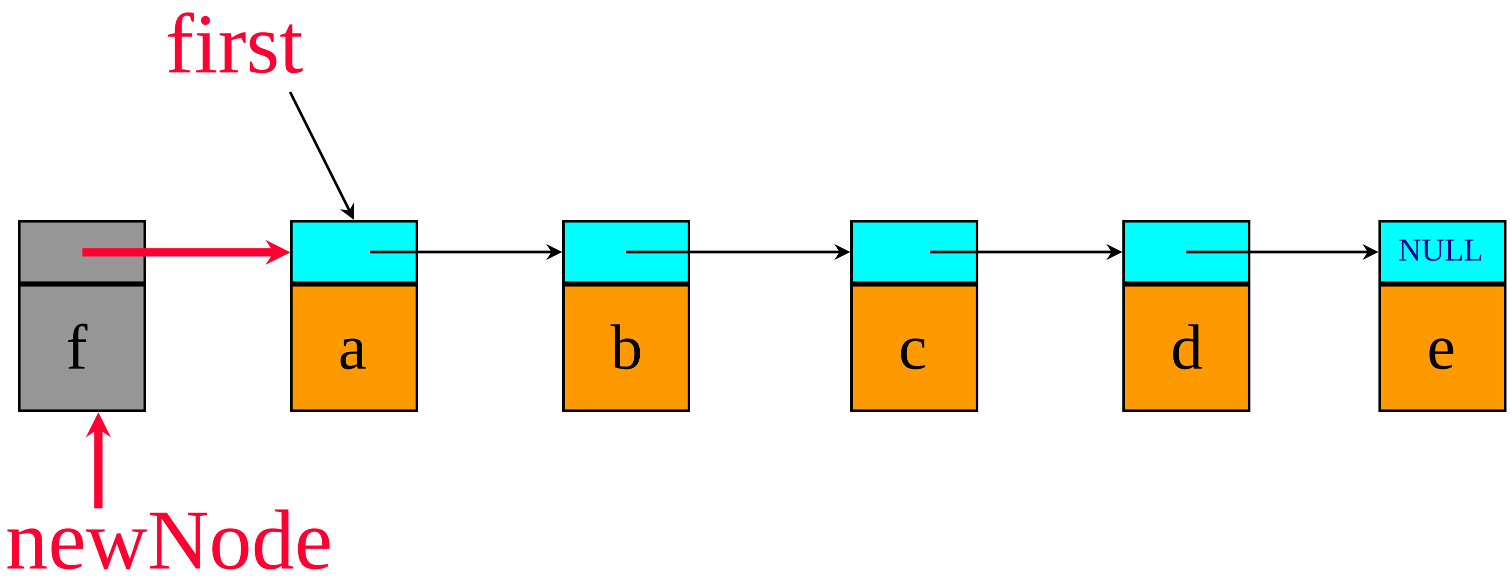


now change pointer in beforeNode

beforeNode–>link = beforeNode–>link–>link;

free(deleteNode);

insert(0,’f’)



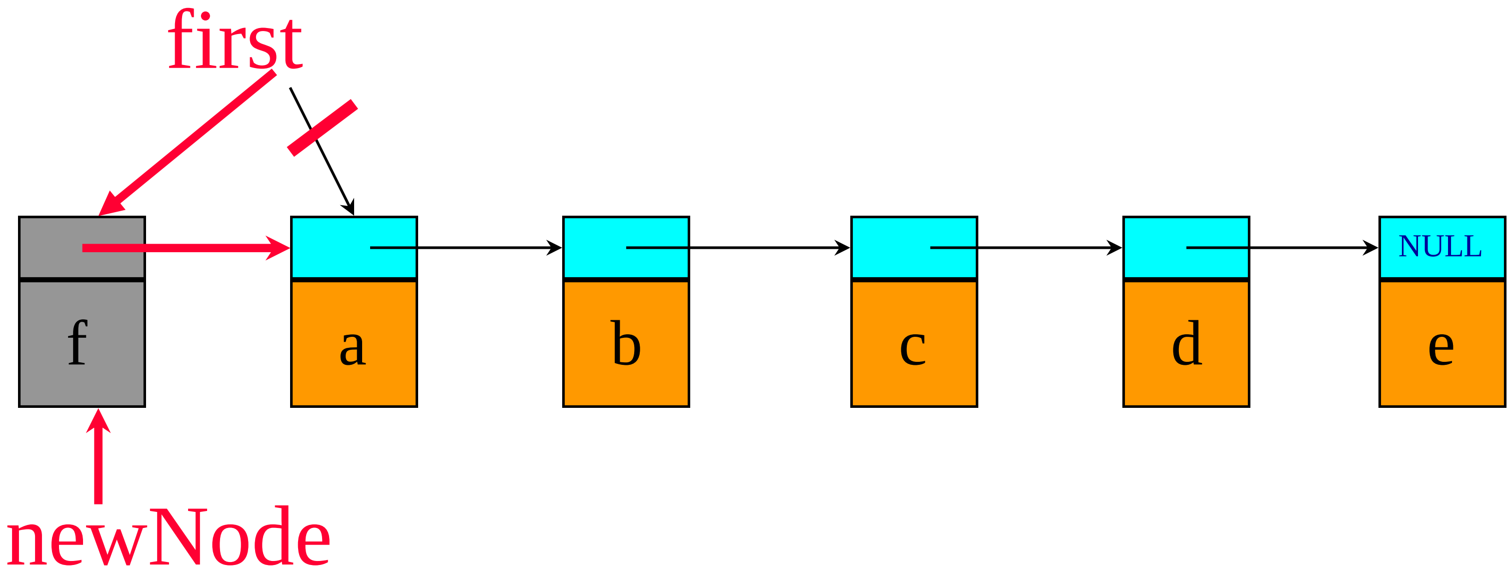
Step 1: get a node, set its data and link fields

MALLOC(newNode, sizeof(*newNode));

newNode->data = ‘f’;

newNode->link = NULL;

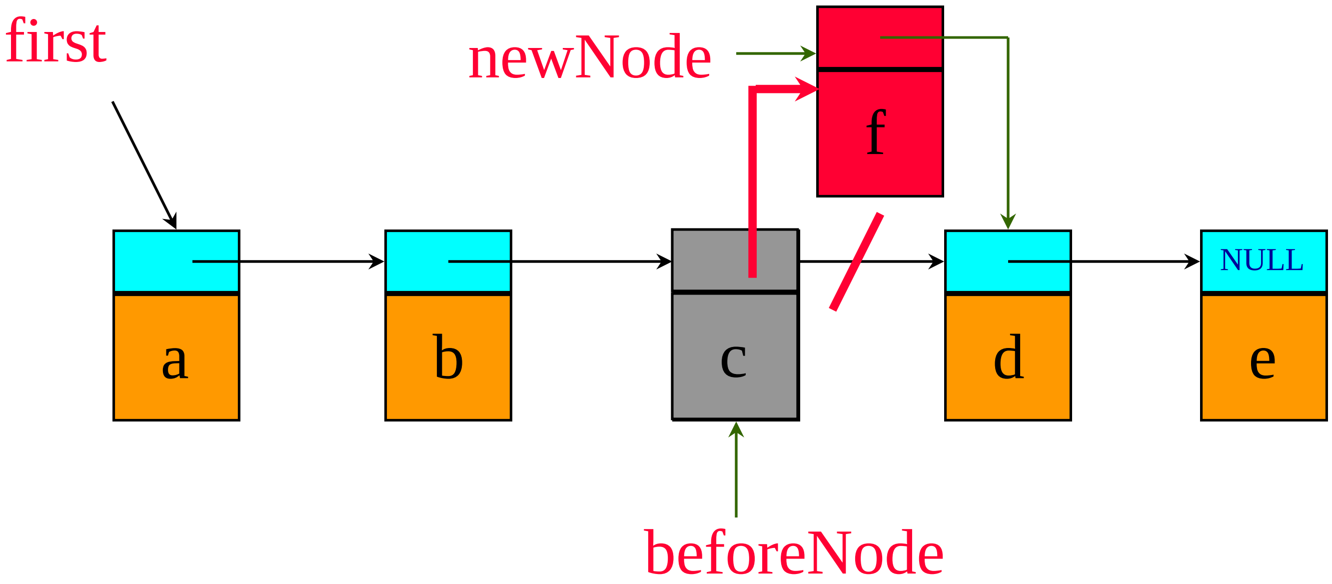
insert(0,'f')



Step 2: update first

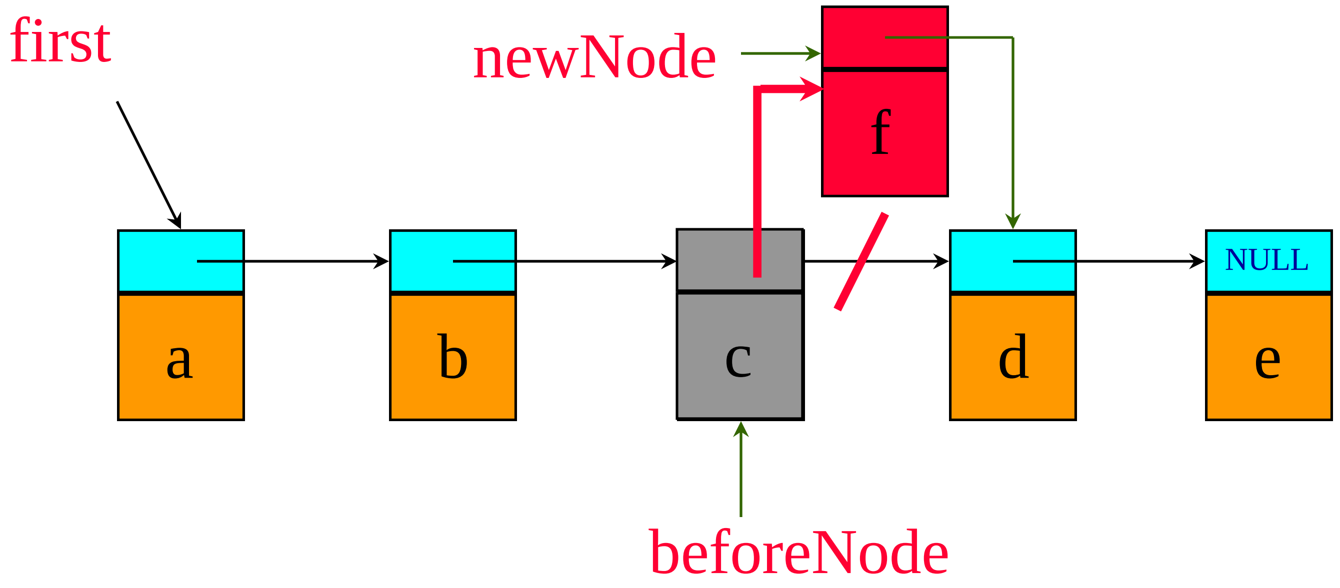
first = newNode;

insert(3,’f’)



- first find node whose index is 2
- next create a new node and set its data and link fields
- finally link beforeNode to newNode

insert(3,'f')



beforeNode = first–>link–>link;

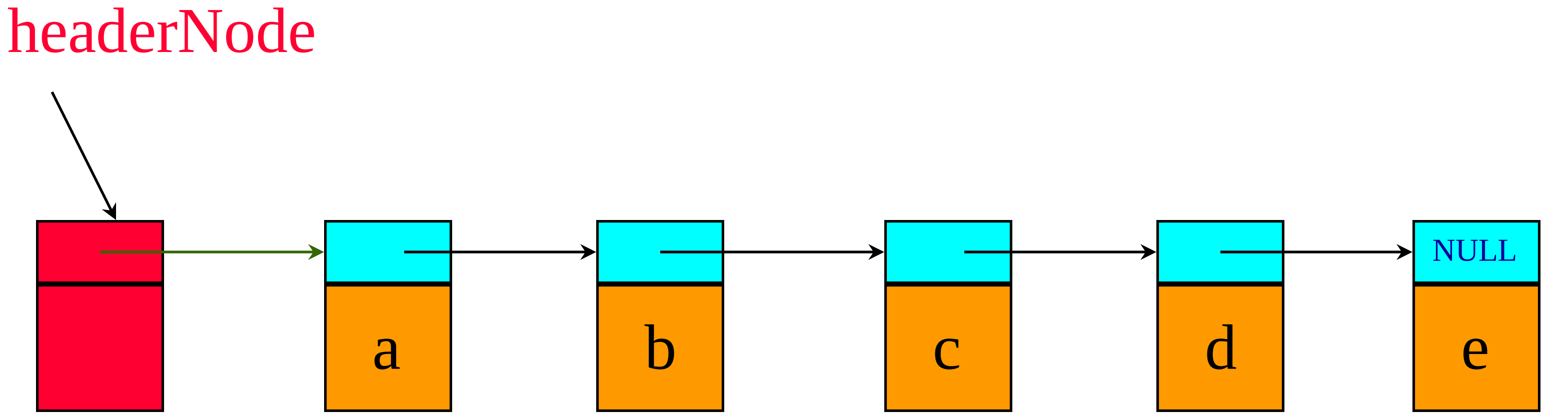
MALLOC(newNode, sizeof(*newNode));

newNode->data = 'f';

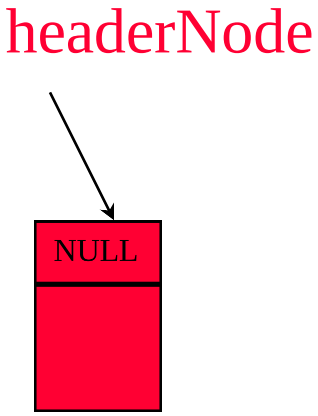
newNode->link = beforeNode->link;

beforeNode–>link = newNode;

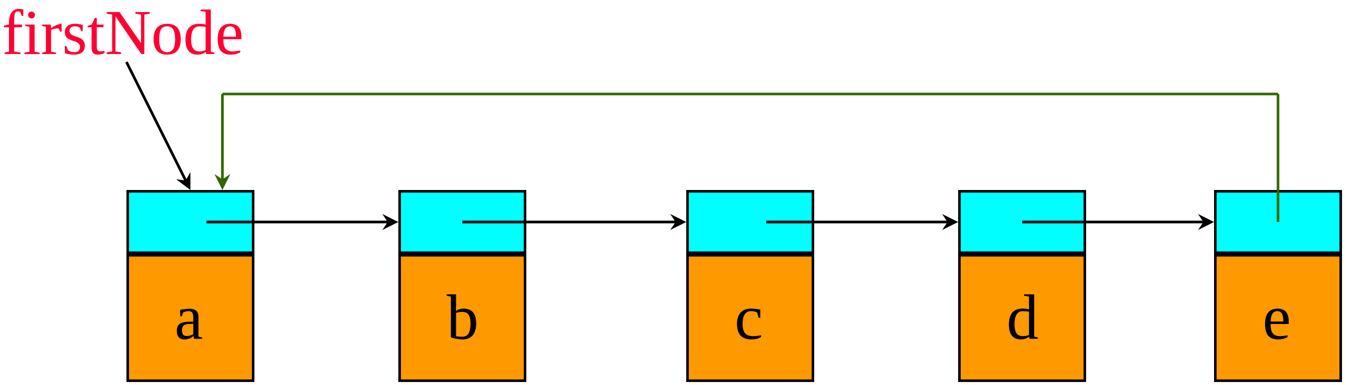
Chain With Header Node



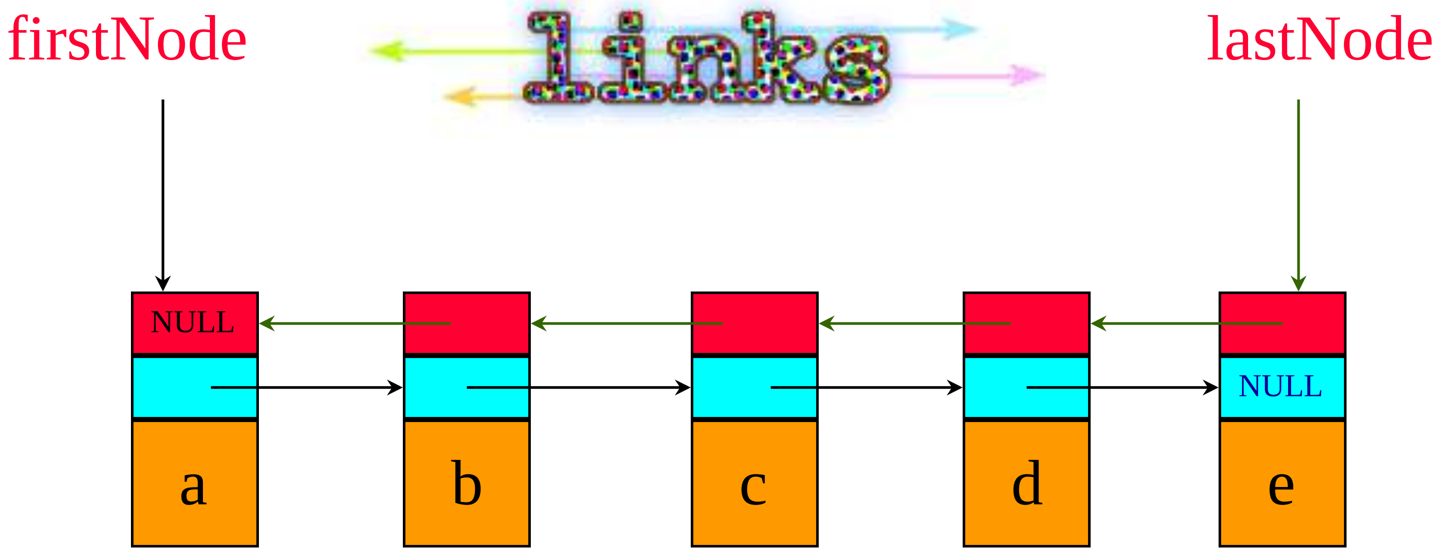
Empty Chain With Header Node



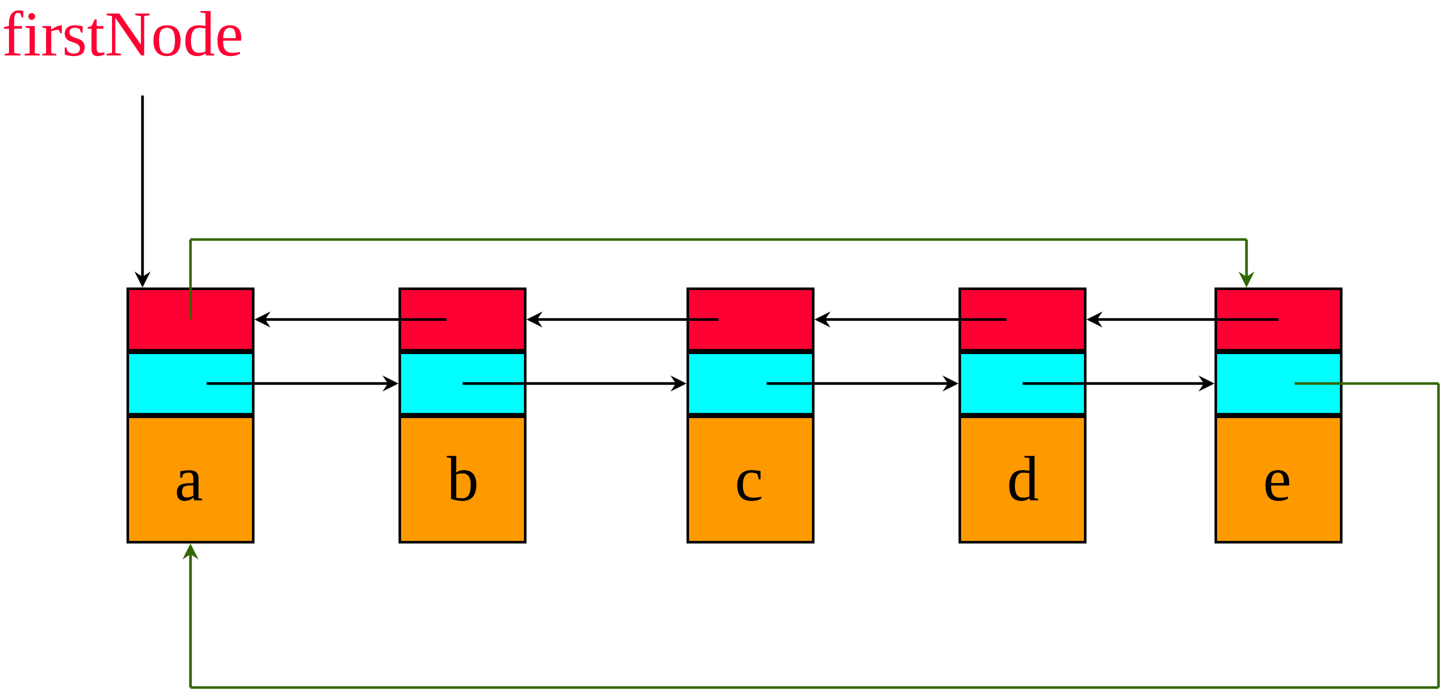
Circular List



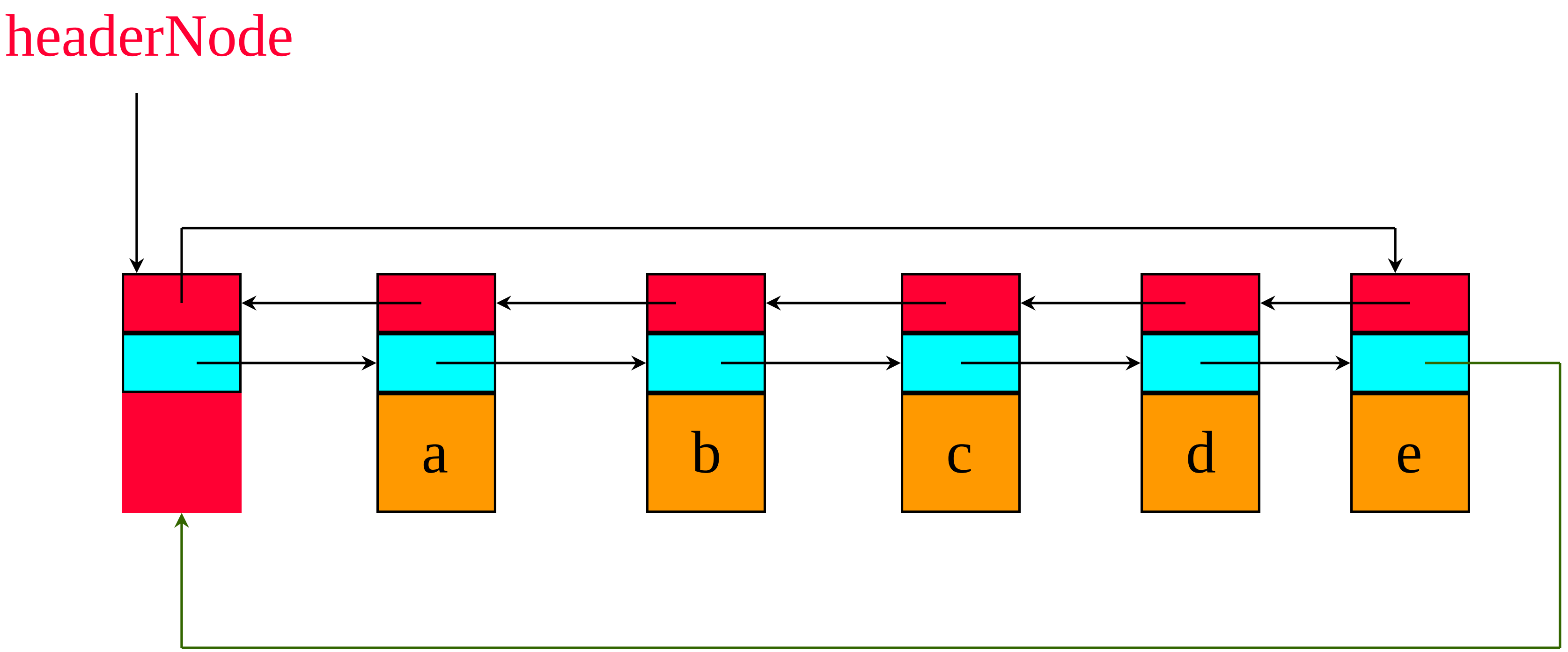
Doubly Linked List



Doubly Linked Circular List



Doubly Linked Circular List With Header Node



Empty Doubly Linked Circular List With Header Node