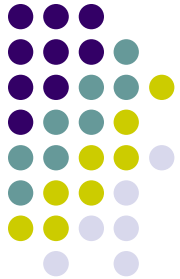
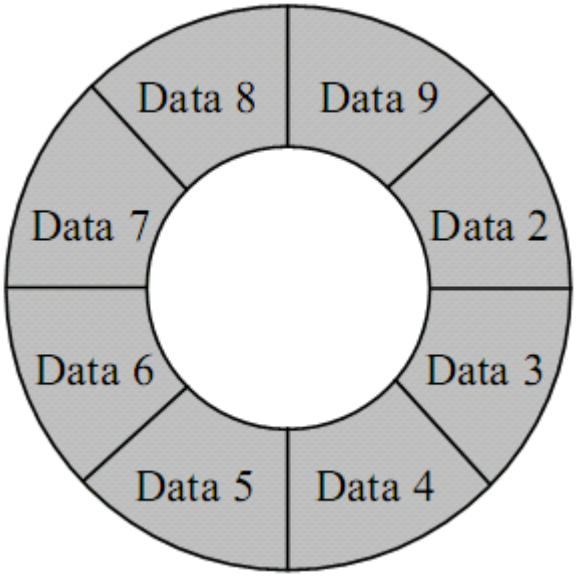
Buffer



- **Buffer** is a data structure which is used to reduce, or even eliminate, the need for synchronization of two different activities. It can be viewed as a piece of memory in which some activity writes/stores results (“producer”) while some other activity is reading results from it (“consumer”). Therefore, buffer is an intermediary between a produces and a consumer.
- Every buffer has a certain number of fields that can contain some values. Number of those fields is buffer size.
- Fields can be of any type.

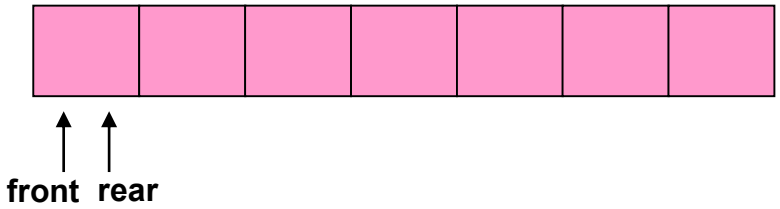
1. Circular buffer

- **Circular/ring buffer** is a data structure that uses a single, fixed-size buffer like it is connected end-to-end, forming a circle/ring.
- It is a FIFO data structure.
- It has two pointers (or indexes) that specify where data can be written (“tail”) and where from it can be read (“head”).
- The advantage of such organization is that there is no need to move elements – only one of the pointers/indexes needs to be updated.
- It can happen that producer overwrites data packages that consumer have not read yet.
  - Either we have to use a bigger (big enough) buffer...
  - Or we need to add some synchronization mechanism to the buffer. Still, synchronization (and wait associated with it) is much rarely needed when where there is not buffer.

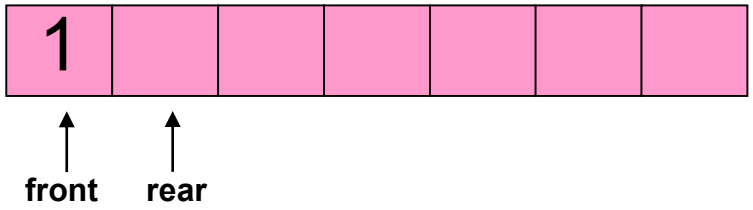


Working with circular buffer

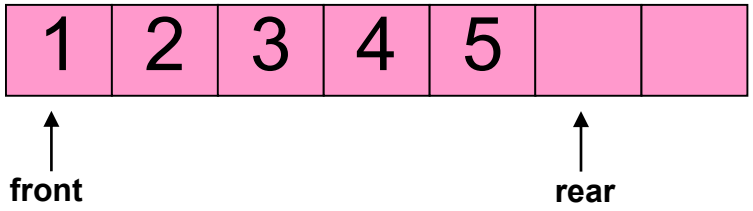
1. **Buffer is empty**



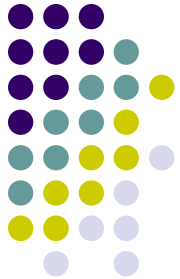
2. **Insert 1**



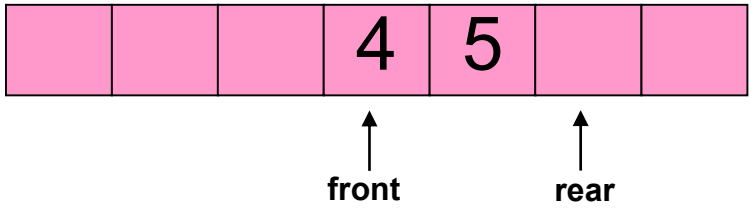
3. **Insert 2, 3, 4 and 5**



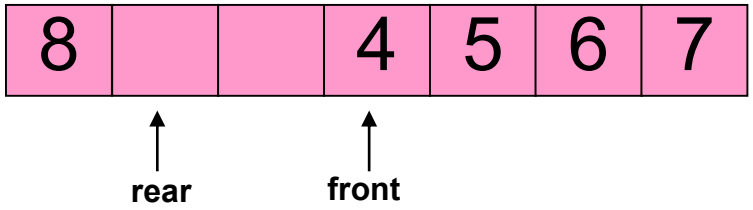
Working with circular buffer



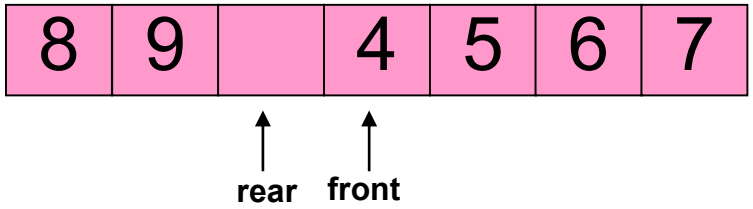
4. Extract 1, 2 and 3



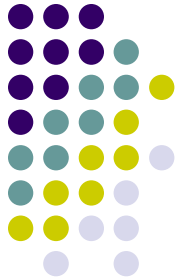
5. Insert 6, 7 and 8



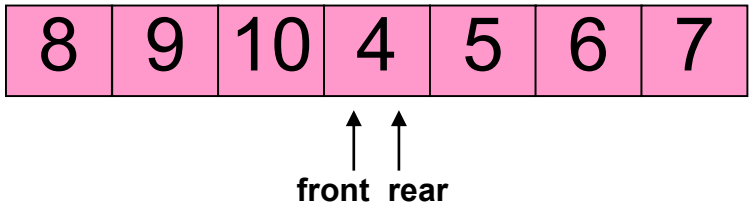
6. Insert 9 => If we need information whether buffer is full or not, this can be the full state.



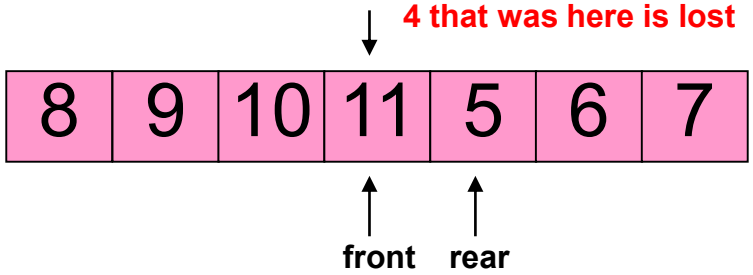
Working with circular buffer



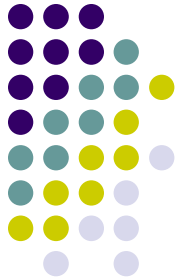
7. Insert 10 => Is buffer full or empty? If an additional variable is introduced to count data then this can be considered to be a full buffer.



8. Insert 11 => Overflow.



Three variants of circular buffer



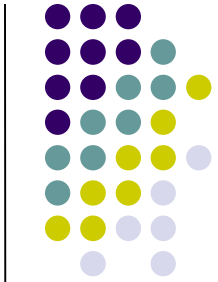
- If we do not need information whether it is full or not:

  Than only two pointers/indexes are needed, and every field in the buffer can be used.
- If we do need info about whether it is full or not:
  - Without additional variable

    One field in the buffer must be kept empty. Simple to implement and cheep to insert/extract values..
  - With additional variable

    One more variable needed. Advantage is the complete usage of the buffer fields, but it increases cost of inserting/extracting because now additional variable has to be updated.

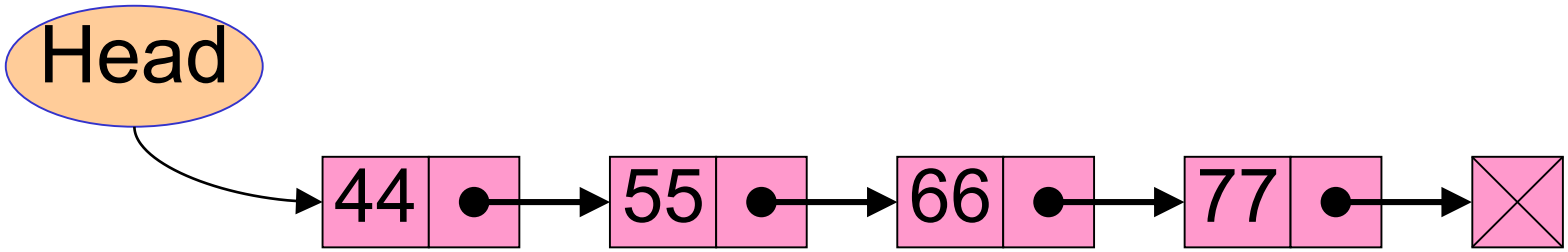
Where are buffers used



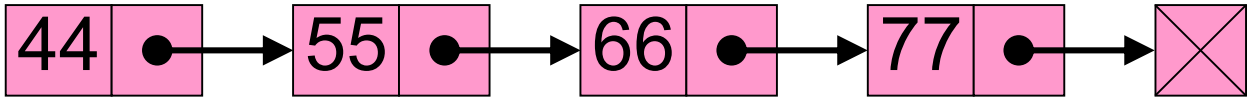
- Buffers, especially circular buffers, are used a lot in system software (communication between devices), and real-time software.
- In real-time software, requirements of real-time, i.e. dead-lines, are most often associated with a particular buffer. E.g. failing to meat the dead-line causes buffer overflow.

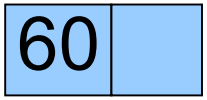
3. Linked list

- A linked list is a data structure in which:
  - Successive nodes (elements) are connected by pointers
  - Last node points to NULL
  - It can grow or shrink during run-time relatively cheaply
  - It can be made just as long as required, free memory is a limit

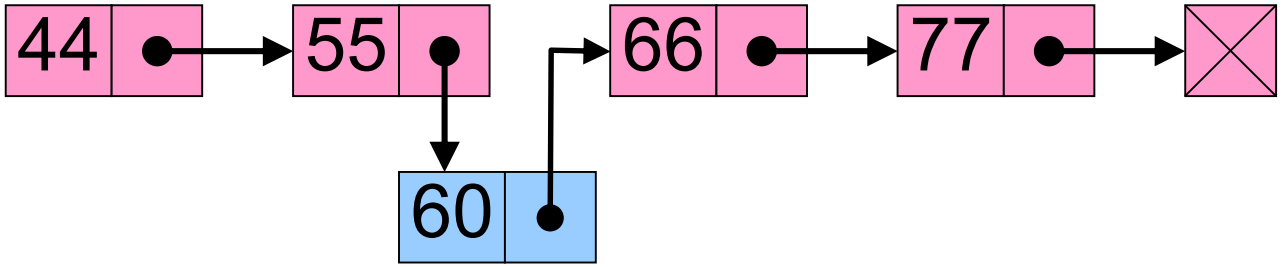


Inserting node



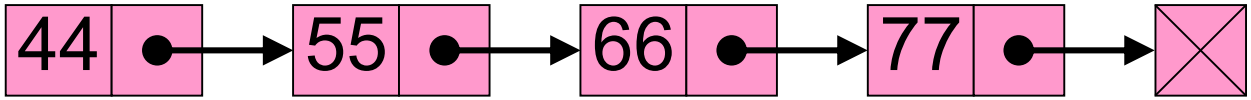


Node to insert

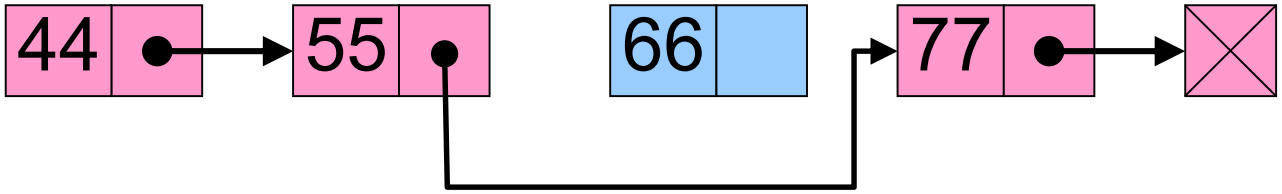


Node to insert

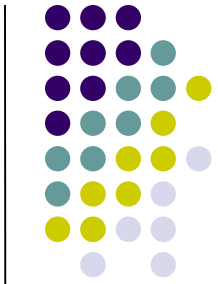
Deleting node



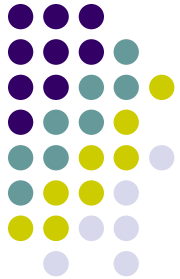
Node to delete



In short



- Insertion:
  - Node is created (new memory is allocated, dynamically) holding the new data.
  - The next pointer of the new node is set to link it to the node which is to follow it in the list.
  - The next pointer of the node which is to precede the new node must be modified to point to it.
- Deletion:
  - The next pointer of the preceding node is made to point to the node following the deleted node.
  - Memory used for deleted node is freed.



Array vs Linked list

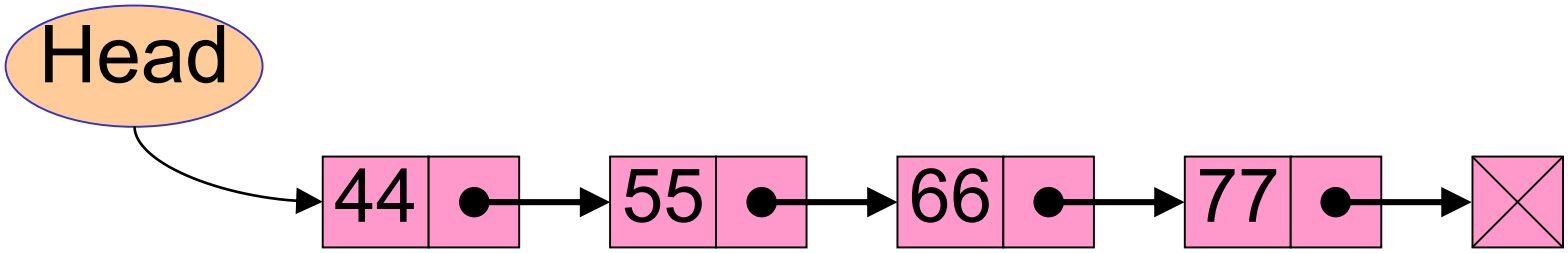
- Arrays are suitable for:
  - Randomly accessing any node (by index)
  - Inserting/deleting node at the end.

- Linked lists are suitable for :
  - Inserting/deleting node anywhere in the list.
  - Applications where sequential access is sufficient.
  - In situations where the number of nodes cannot be predicted beforehand, or when it is needed to work with a lot of elements (Question: what is the limit of array size?)

Types of list

Singly-linked lists

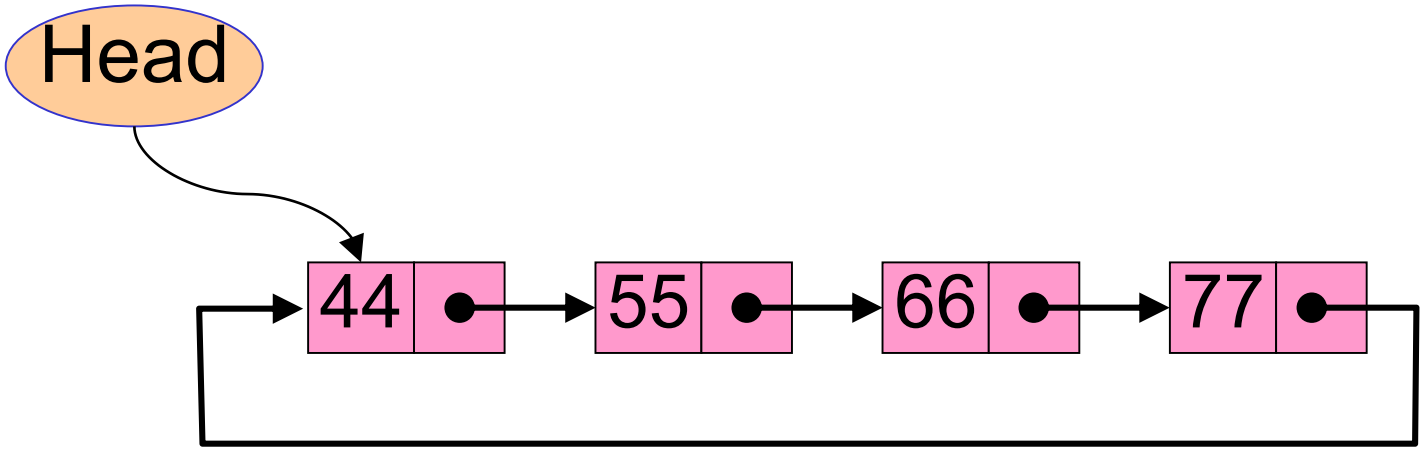
One pointer per node. It points to the next node. The last one points to NULL.



Types of list

Circular singly-linked lists

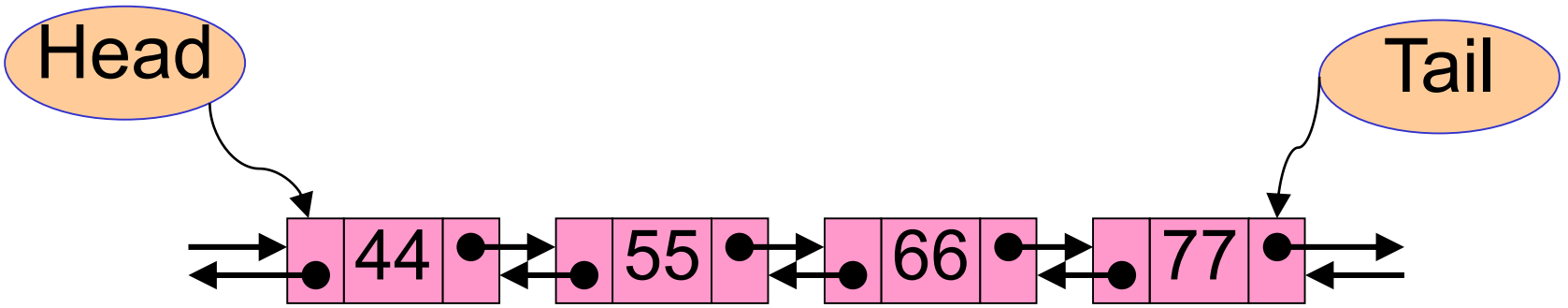
The pointer from the last node in the list points back to the first node.



Types of list

Doubly linked list

- Two pointers per element – one points to the next, and one to the previous node.
- The list can be traversed either forward or backward.
- Usually two pointers are maintained to keep track of the list, **head** and **tail**.



Implementing a list

- For list it is enough to have a pointer that points to the first node. If it points to NULL, the list is empty.
- Creating the first element

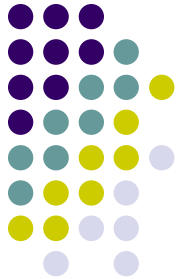
```
struct node
{
   int    id;
   char   name[25];
   int    age;
   struct node* next;
};
```

```
struct person {
  int id;
  char name[25];
  int age;
};

struct node {
  struct person data;
  struct node* next;
};
```

```
head = (struct node*)malloc(sizeof(struct node));
```

- For each node, the following needs to be done:
  - **Allocating memory**
  - **Setting up field values**
  - **Updating node pointers**

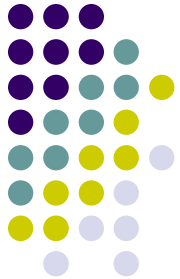
Traversing the list



- Simple operation that does the following:
  - Start from the head (or tail, in case of doubly linked list)
  - Follow the links, i.e. pointers to next node (or previous node, if traversing doubly linked list backwards)
  - Stop when the pointer is NULL

    (When to stop in case of circular list?)

Inserting node in the list

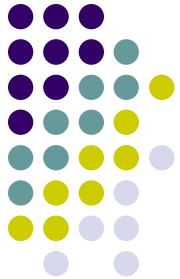


- **Inserting node at the beginning**
  - New node points to the previously first element (what the head points to).
  - Head is made to point to the new node.
- **Inserting node at the end**
  - Last node now points to the new node.
  - New node points to NULL.
- **Inserting in the middle**
  - New node points to the next node.
  - Previous node now points to the new node.

For inserting a new node in a linked list, we need to know after which element (or head) we want to insert it.

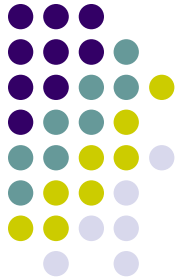
Removing element is analogous to inserting.

Inserting node in the list



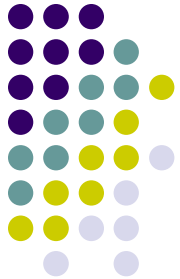
```
struct node {
  struct person data;
  struct node* next;
};
struct node* List;

void addEleInFront(struct node** head, struct node* element) {
  element->next = *head;
  *head = element;
}
void addEle(struct node* place, struct node* element) {
  element->next = place->next;
  place->next = element;
}
void addEleAtEnd(struct node* last, struct node* element) {
  element->next = NULL;
  last->next = element;
}
```

Inserting node in the list



```
struct node {
  struct person data;
  struct node* next;
};
struct node* List;

void addEleInFront(struct node** head, struct node* element) {
  element->next = *head;
  *head = element;
}
void addEle(struct node* place, struct node* element) {
  element->next = place->next;
  place->next = element;
}
void addEleAtEnd(struct node* last, struct node* element) {
  element->next = NULL; // last->next == NULL
  last->next = element;
}
```

Inserting node in the list

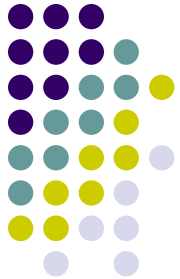


```
struct node {
  struct person data;
  struct node* next;
};
struct node* List;

void addEleInFront(struct node** head, struct node* element) {
  element->next = *head;
  *head = element;
}
void addEle(struct node* place, struct node* element) {
  element->next = place->next;
  place->next = element;
}

addEleInFront(&List, ele);
addEle(pos, ele);
```

Inserting node in the list

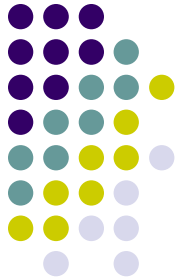
```
struct node {
  struct person data;
  struct node* next;
};
struct node* List;

void addEleInFront(struct node** head, struct node* element) {
  element->next = *head;
  *head = element;
}
void addEle(struct node** nextAddr, struct node* element) {
  element->next = *nextAddr;
  *nextAddr = element;
}

addEle(&List, ele);
addEle(&(pos->next), ele);
```

Inserting node in the list

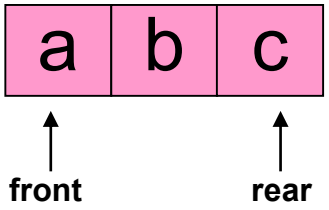


```
struct node {
  struct person data;
  struct node* next;
};
struct node List; // <-

void addEleInFront(struct node** head, struct node* element) {
  element->next = *head;
  *head = element;
}
void addEle(struct node* place, struct node* element) {
  element->next = place->next;
  place->next = element;
}

addEle(&List, ele);
addEle(pos, ele);
```

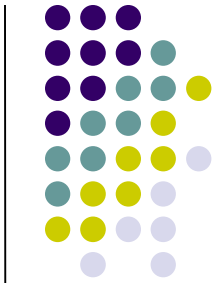
4. Queue



- Queue is essentially a data structure that is accessed in FIFO manner.
- Queue can be implemented in different ways.
- For something to be a queue is it enough to able to hold multiple elements of the same type and that they can be accessed in FIFO manner.
- In theory, queue is of infinite size, i.e. it can hold any number of elements. In practice that is impossible, so this comes down to the fact that queue overflow is not defined.
- On the other hand, we must be able to query whether queue is empty.



Queue – Array Implementation

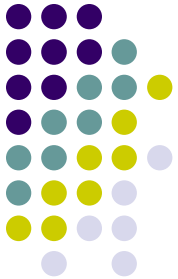


- That is a circular buffer!

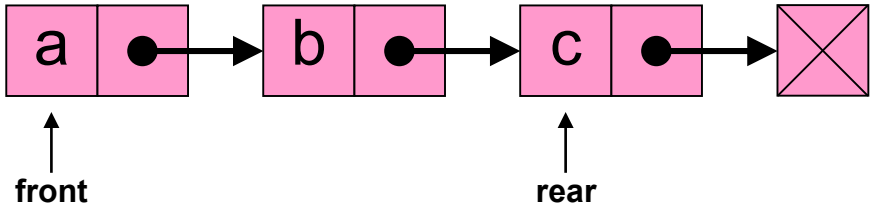
  Only we need information whether it is full so we have to use those implementations that provide that information.

  And that is all. ☺

Queue - Linked-List Implementation

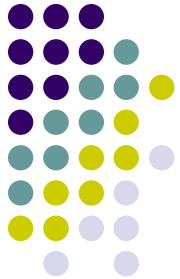


- Singly linked list is good enough for implementing a queue.
- Elements are added at the end, and removed from the beginning.
- The only issue is the fact that we need the access to the last element in order to add nodes at the end.
- That is why it is good to introduce a new pointer that will point to the last element.

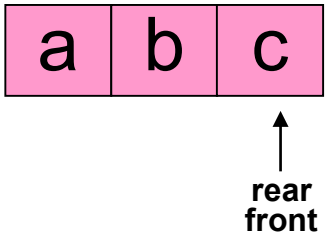


- However, that increases the cost of adding node, because we need to update that pointer as well

5. Stack



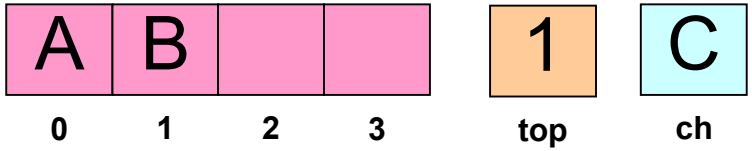
- Stack is, in a way, opposite of queue. Data has to be accessed in LIFO manner.
- Stack too can be implemented in different ways.
- Stack too, in theory, can hold any number of elements. Again, it comes down to the fact that stack overflow is not defined either.
- For stack too we must be able to query whether it is empty.



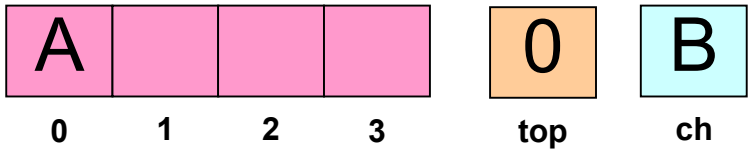
Stack – Array Implementation

We also need a variable which will tell us where is the top of the stack:

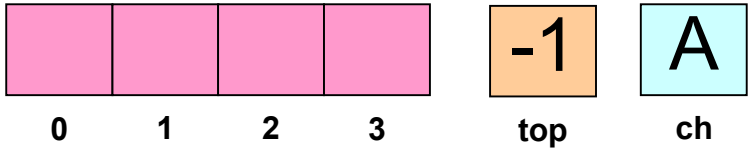
```
ch = Pop(stack)
```




```
ch = Pop(stack)
```

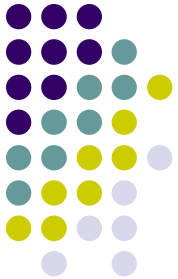



```
ch = Pop(stack)
```



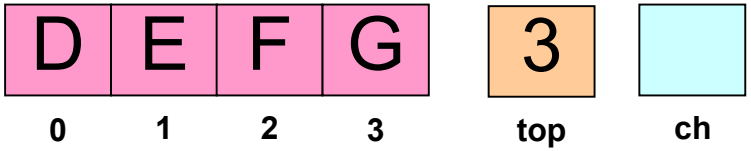
Stack is empty when top is **-1**.

Stack – Array Implementation



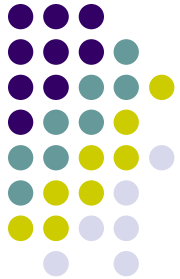
What happens if we exceed maximal stack size? Example:

```
Push(stack, 'D')
Push(stack, 'E')
Push(stack, 'F')
Push(stack, 'G')
```



And then we try this: **`Push(stack, 'H')`**

Stack – Linked List Implementation



Singly linked list is good enough for implementing a stack.

Where the **top** of the stack should be, **at the beginning or end** of the list?