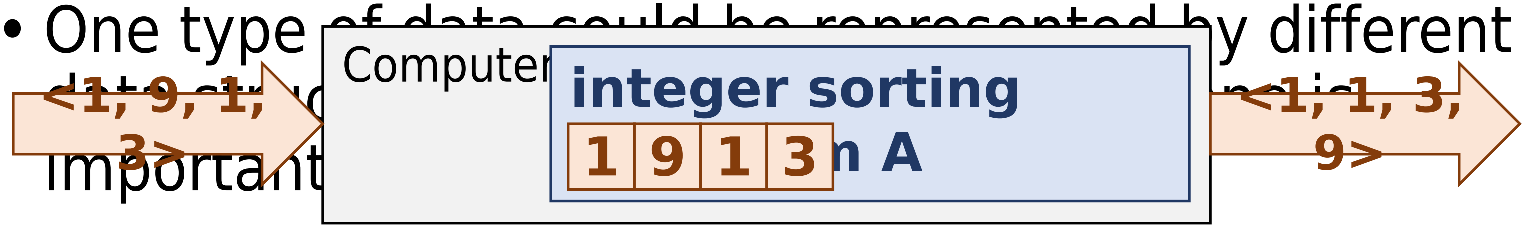
Basic Data Structures

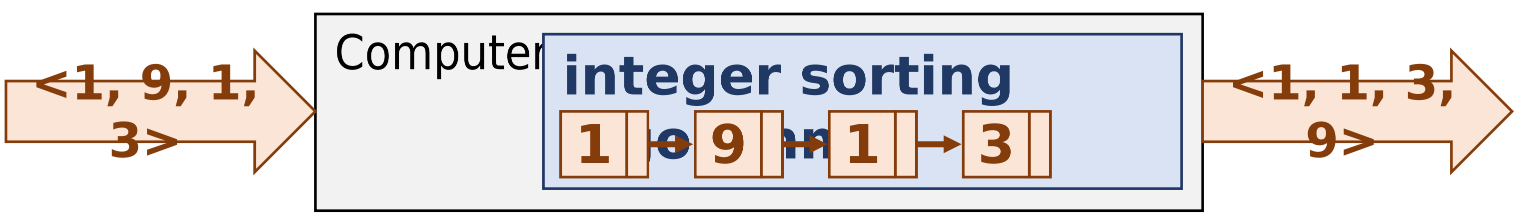
Data Structures and Algorithms

Nanjing University, Fall 2021
郑朝栋

What is a “data structure”?

- A **data structure** is a way to **store and organize data** in order to facilitate **access** and **modifications**.
- E.g., *array* and *linked list*.
- Different types of data demand different data structures.
- One type of data could be represented by different





Abstract Data Type (ADT)

- A data structure usually provides an **interface**.
  - Often, the interface is also called an **abstract data type (ADT)**.
  - An ADT specifies what a data structure "can do" and "should do", but *not* "how to do" them.
- ADT: List, which supports get, set, add, remove, …
  Data structure: ArrayList, LinkedList, …
- An ADT is a logical description, and a data structure is a concrete implementation.
  - Similar to .h file and .cpp file.
  - Different data structures can implement same ADT. (Why bother?)

The Queue ADT

The Queue ADT represents a collection of items to which we can add items and remove the next item.

- Add(x): add x to the queue.
- Remove(): remove the next item y from queue, return y.

The *queuing discipline* decides which item to be removed.

FIFO Queue

The Queue ADT represents a collection of items to which we can add items and remove the next item.

- Add(x): add x to the queue.
- Remove(): remove the next item y from queue, return y.

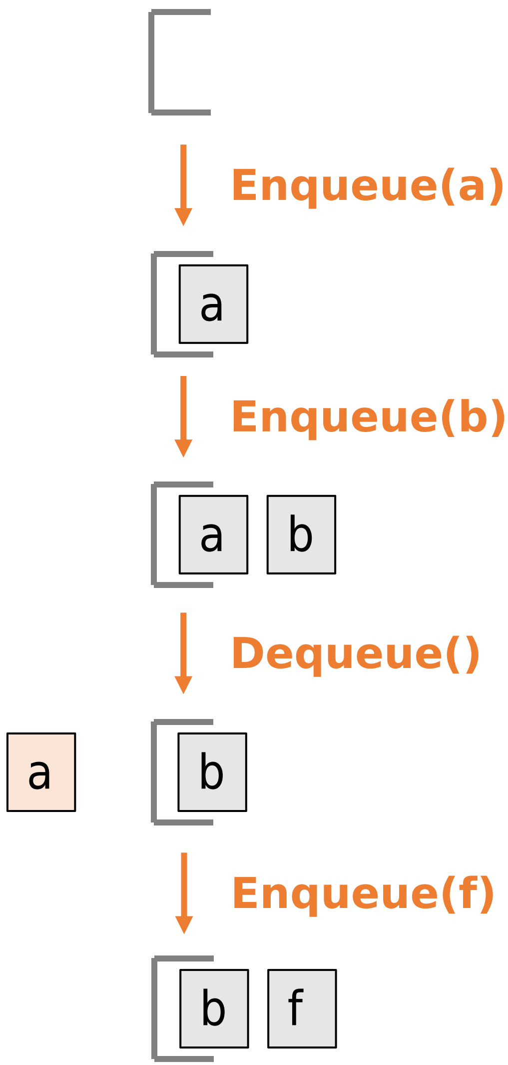
The **first-in-first-out (FIFO)** queuing discipline: items are removed in the same order they are added.

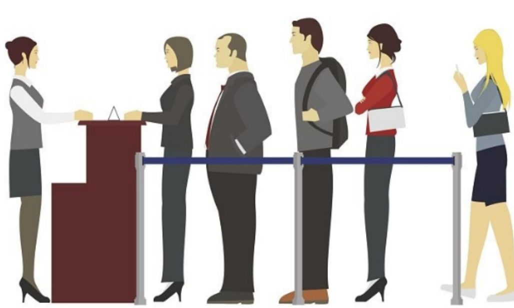
FIFO Queue:

- Add(x) or **Enqueue(x)**: add x to the end of the queue.
- Remove() or **Dequeue()**: remove the first item from the queue.

Enqueue(x)

Dequeue()


Enqueue(a)
a
Enqueue(b)
a
b
Dequeue()
a
b
Enqueue(f)
b
f



LIFO Queue: Stack

The Queue ADT represents a collection of items to which we can add items and remove the next item.

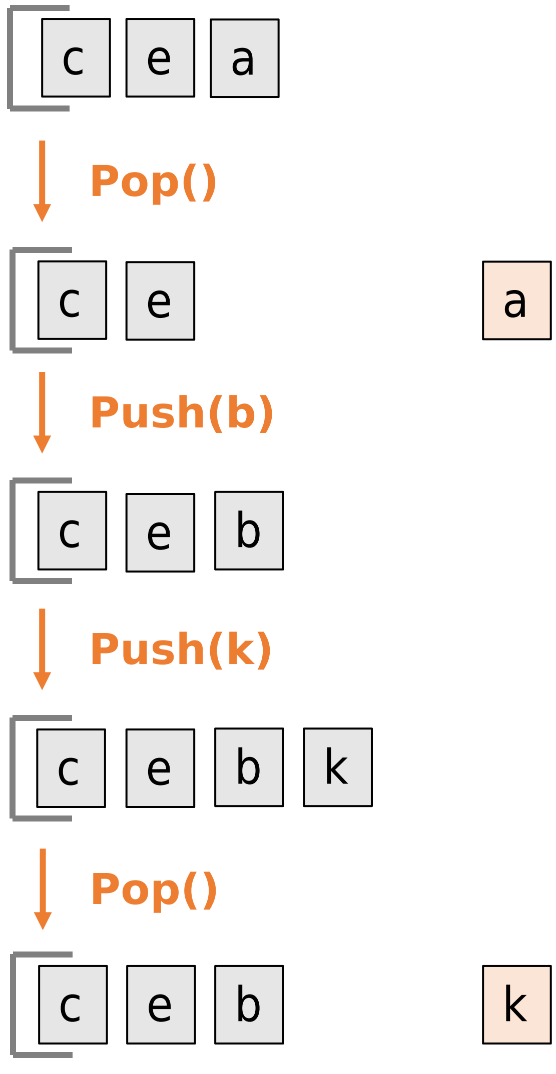
- Add(x): add x to the queue.
- Remove(): remove the next item y from queue, return y.

The **last-in-first-out (LIFO)** queuing discipline:
the most recently added item is the next one removed.

Stack (LIFO Queue):

- Add(x) or **Push(x)**:
  add x to the top of the stack.
- Remove() or **Pop()**:
  remove the item at the top of the stack.

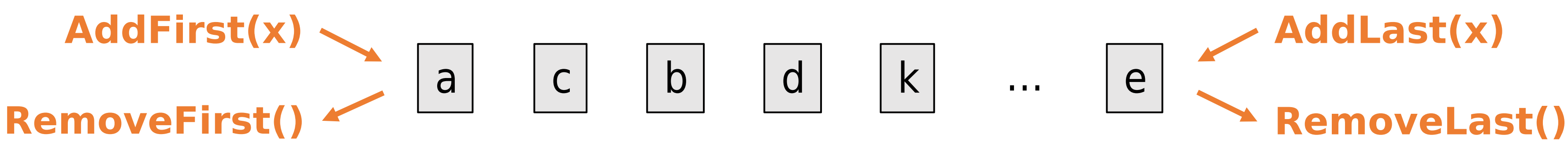
Pop() Push(x)


c e a
Pop()
c e
a
Push(b)
c e b
Push(k)
c e b k
Pop()
c e b
k

The Deque ADT

The **Deque** (double-ended queue) ADT represents a sequence of items with a front and a back.

- AddFirst(x), AddLast(x), RemoveFirst(), RemoveLast().



Deque can implement FIFO Queue:

- Enqueue(x) is AddLast(x), Dequeue() is RemoveFirst().

Deque can implement Stack (LIFO Queue):

- Push(x) is AddLast(x), Pop() is RemoveLast().

The List ADT

1	2	3	4	5	…	n
a	c	b	d	k	…	e

- A **List** is a sequence of items $x_1, x_2, \cdots, x_n$
- The List interface supports the following operations:
  - Size(): return n, the length of the list
  - Get(i): return x_i
  - Set(i,x): set $x_i = x$
  - Add(i,x):
    set $x_{j+1} = x_j$ for $n \geq j \geq i$, set $x_i = x$, increase list size by 1
  - Remove(i):
    set $x_j = x_{j+1}$ for $i \leq j \leq n-1$, decrease list size by 1

The List ADT

1	2	3	4	5	…	n
a	c	b	d	k	…	e

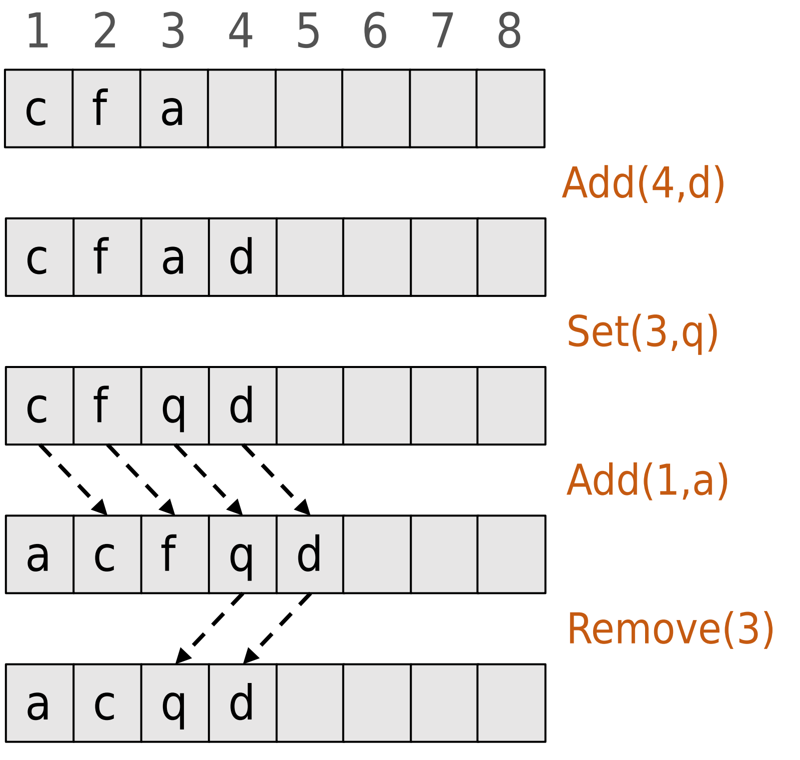
- The List interface supports the following operations:
  - Size(): return n, the length of the list
  - Get(i): return x_i
  - Set(i,x): set $x_i = x$
  - Add(i,x):
    set $x_{j+1} = x_j$ for $n \geq j \geq i$, set $x_i = x$, increase list size by 1
  - Remove(i):
    set $x_j = x_{j+1}$ for $i \leq j \leq n-1$, decrease list size by 1
- List can implement Deque:
  - AddFirst(x) is Add(1,x), AddLast(x) is Add(Size()+1,x)
  - RemoveFirst() is Remove(1), RemoveLast() is Remove(Size())

Using array to implement List:

ArrayList data structure

The List interface supports the following operations:

- **Size():** always $\Theta(1)$
  return n, the length of the list
- **Get(i):** always $\Theta(1)$
  return x_i
- **Set(i,x):** always $\Theta(1)$
  set $x_i = x$
- **Add(i,x):** $\Theta(1)$ to $\Theta(n)$
  set $x_{j+1} = x_j$ for $n \geq j \geq i$,
  set $x_i = x$, increase n by 1
- **Remove(i):** $\Theta(1)$ to $\Theta(n)$
  set $x_j = x_{j+1}$ for $i \leq j \leq n-1$,
  decrease n by 1



- Queries and updates are fast.
- Modifications are fast at “end”, but slow at “front” or “middle”.

Using array to implement List:

ArrayList data structure

The List interface supports the following operations:

- **Size():** always $\Theta(1)$
  return n, the length of the list
- **Get(i):** always $\Theta(1)$
  return x_i
- **Set(i,x):** always $\Theta(1)$
  set $x_i = x$
- **Add(i,x):** $\Theta(1)$ to $\Theta(n)$
  set $x_{j+1} = x_j$ for $n \geq j \geq i$,
  set $x_i = x$, increase n by 1
- **Remove(i):** $\Theta(1)$ to $\Theta(n)$
  set $x_j = x_{j+1}$ for $i \leq j \leq n-1$,
  decrease n by 1

Q: Is ArrayList good for Stack?
A: Yes. (Push and Pop are fast.)

Q: Is ArrayList good for FIFO Queue?
A: No. (Enqueue can be slow.)

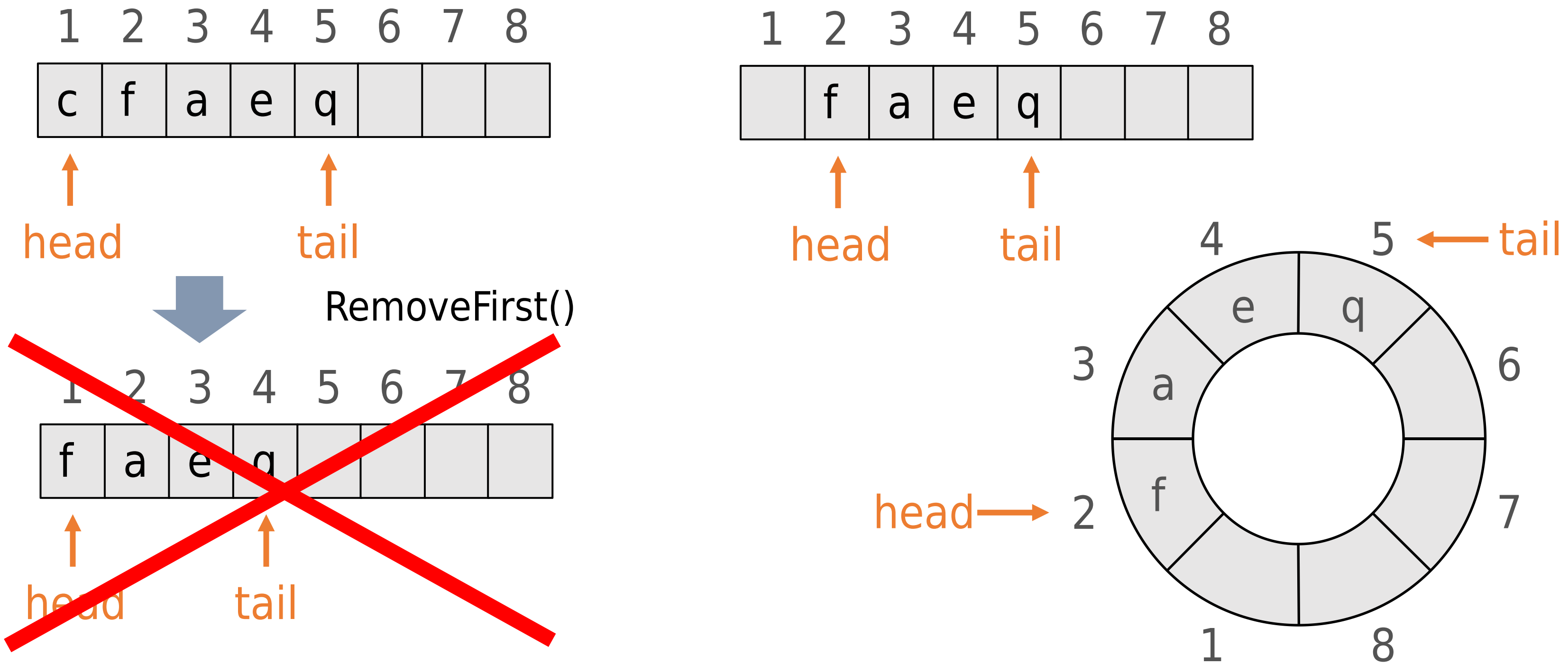
Q: Is ArrayList good for Deque?
A: No.

- Queries and updates are fast.
- Modifications are fast at “end”, but slow at “front” or “middle”.

Using circular array to implement Deque:
ArrayDeque data structure

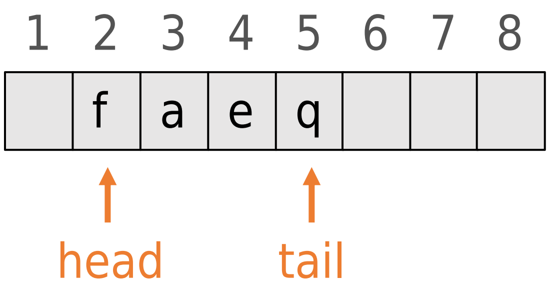
Using simple array to implement List:

- Queries and updates are fast.
- Modifications are fast at “end”, but slow at “front” or “middle”.
- ArrayList is good for Stack, but not FIFO Queue or Deque.



Using circular array to implement Deque:
ArrayDeque data structure

Maintain head and tail:
- AddFirst and RemoveFirst: move head.
- AddLast and RemoveLast: move tail.
- Use modular arithmetic to "wrap around" at both ends.



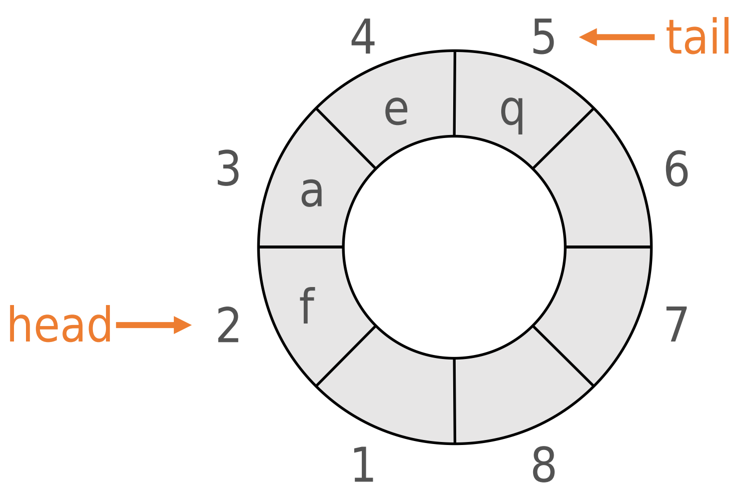
AddLast(x):
tail=(tail%N)+1
A[tail]=x

RemoveFirst():
head=(head%N)+1

AddFirst(x):
head=(head==1)?N:(head-1)
A[head]=x

RemoveLast(x):
tail=(tail==1)?N:(tail-1)

all in $O(1)$

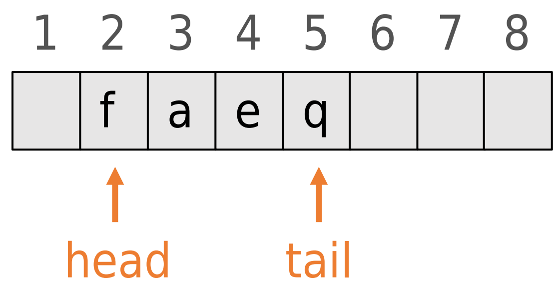


Using circular array to implement Deque:

ArrayDeque data structure

Maintain head and tail:

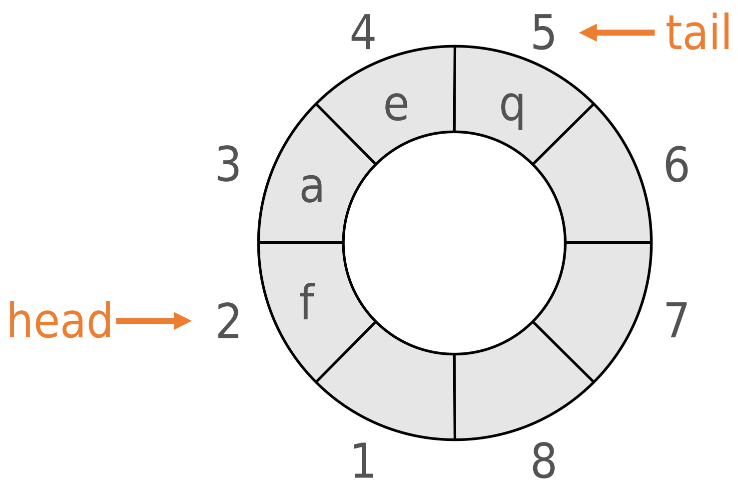
- AddFirst and RemoveFirst: move head.
- AddLast and RemoveLast: move tail.
- Use modular arithmetic to "wrap around" at both ends.



- Queries and updates are fast.
- Modifications are fast at "front" and "end" (i.e., head and tail), but still slow at "middle".

- ArrayDeque is good for Stack, FIFO Queue, and Deque; but can be slow for some List operations.

- Capacity of array is also a problem!!!



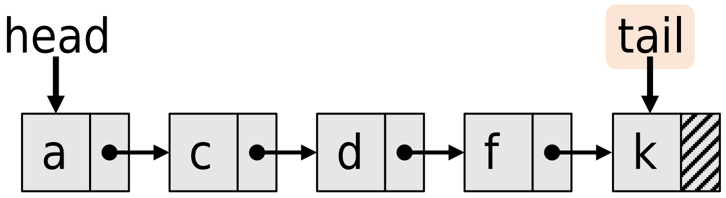
Using linked list to implement List:

LinkedList data structure

The List interface supports the following operations:

- **Size():** always $\Theta(1)$
  return n, the length of the list
- **Get(i):** $\Theta(1)$ to $\Theta(n)$
  return x_i
- **Set(i,x):** $\Theta(1)$ to $\Theta(n)$
  set $x_i = x$
- **Add(i,x):** $\Theta(1)$ to $\Theta(n)$
  set $x_{j+1} = x_j$ for $n \geq j \geq i$,
  set $x_i = x$, increase n by 1
- **Remove(i):** $\Theta(1)$ to $\Theta(n)$
  set $x_j = x_{j+1}$ for $i \leq j \leq n-1$,
  decrease n by 1

Traversing backwards from tail is not efficient.



Q: Is LinkedList good for Stack?
A: Yes. (Push and Pop at head are fast.)

Q: Is LinkedList good for FIFO Queue?
A: Yes. (Enqueue and Dequeue are fast.)

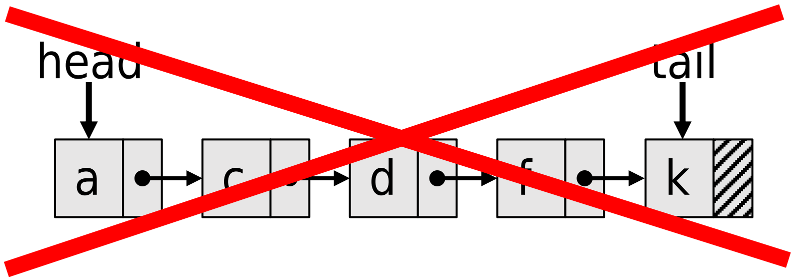
Q: Is LinkedList good for Deque?
A: No. (RemoveLast can be slow.)

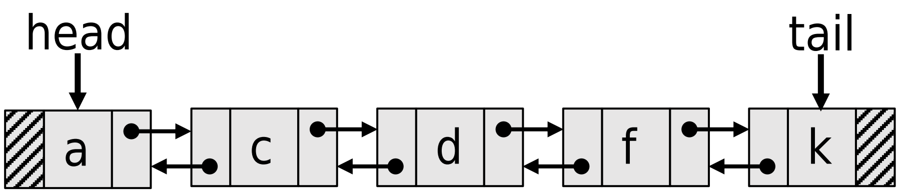
Using doubly-linked list to implement List:

DLinkedList data structure

The List interface supports the following operations:

- **Size():** always $\Theta(1)$
  return n, the length of the list
- **Get(i):** $\Theta(1)$ to $\Theta(n)$
  return x_i
- **Set(i,x):** $\Theta(1)$ to $\Theta(n)$
  set $x_i = x$
- **Add(i,x):** $\Theta(1)$ to $\Theta(n)$
  set $x_{j+1} = x_j$ for $n \geq j \geq i$,
  set $x_i = x$, increase n by 1
- **Remove(i):** $\Theta(1)$ to $\Theta(n)$
  set $x_j = x_{j+1}$ for $i \leq j \leq n-1$,
  decrease n by 1





DLinkedList is good for Stack, FIFO Queue, and Deque; but can be slow for some List operations.

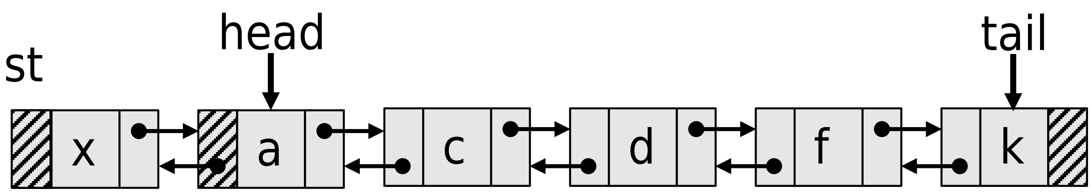
Using doubly-linked list to implement List:

DLinkedList data structure

The List interface supports the following operations:

- **Size():** always $\Theta(1)$
  return n, the length of the list
- **Get(i):** $\Theta(1)$ to $\Theta(n)$
  return x_i
- **Set(i,x):** $\Theta(1)$ to $\Theta(n)$
  set $x_i = x$
- **Add(i,x):** $\Theta(1)$ to $\Theta(n)$
  set $x_{j+1} = x_j$ for $n \geq j \geq i$,
  set $x_i = x$, increase n by 1
- **Remove(i):** $\Theta(1)$ to $\Theta(n)$
  set $x_j = x_{j+1}$ for $i \leq j \leq n-1$,
  decrease n by 1

DLinkedList is good for Stack, FIFO Queue, and Deque; but can be slow for some List operations.



What if head==NULL?

<u>AddFirst(x):</u>
```
x.next=head
head->prev=&x
head=&x
x.prev=NULL
```

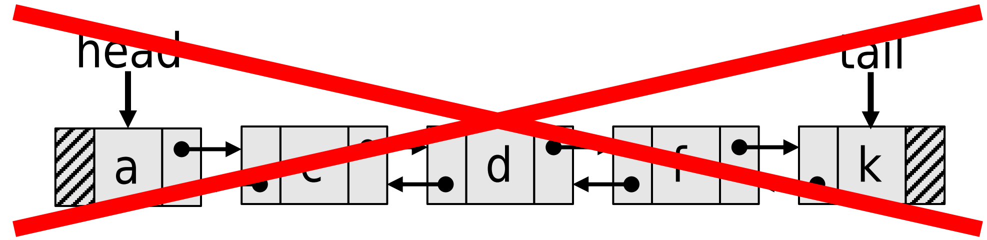
What about tail?

<u>AddFirst(x):</u>
```
x.next=head
if (head!=NULL)
  head->prev=&x
head=&x
x.prev=NULL
```

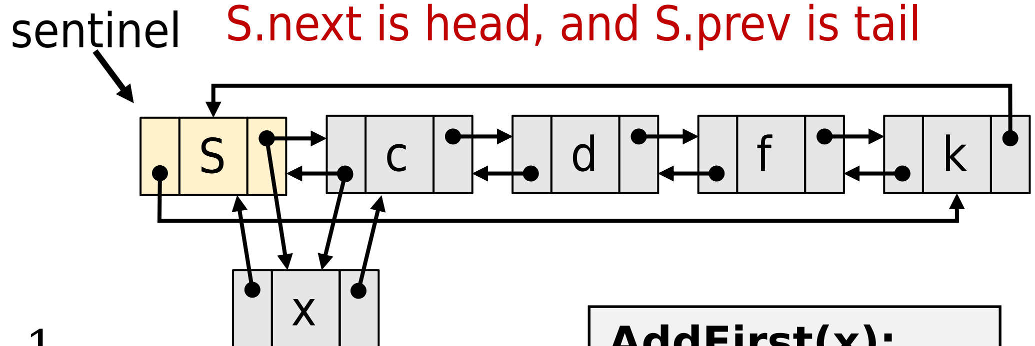
Using doubly-linked list to implement List:
DLinkedList data structure

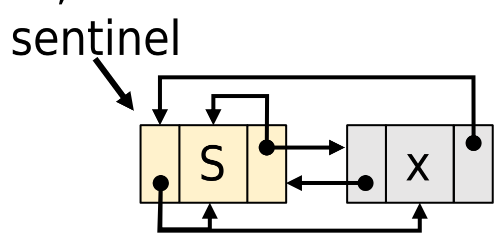
The List interface supports the following operations:

- **Size():** always $\Theta(1)$
  return n, the length of the list
- **Get(i):** $\Theta(1)$ to $\Theta(n)$
  return x_i
- **Set(i,x):** $\Theta(1)$ to $\Theta(n)$
  set $x_i = x$
- **Add(i,x):** $\Theta(1)$ to $\Theta(n)$
  set $x_{j+1} = x_j$ for $n \geq j \geq i$,
  set $x_i = x$, increase n by 1
- **Remove(i):** $\Theta(1)$ to $\Theta(n)$
  set $x_j = x_{j+1}$ for $i \leq j \leq n-1$,
  decrease n by 1



S.next is head, and S.prev is tail





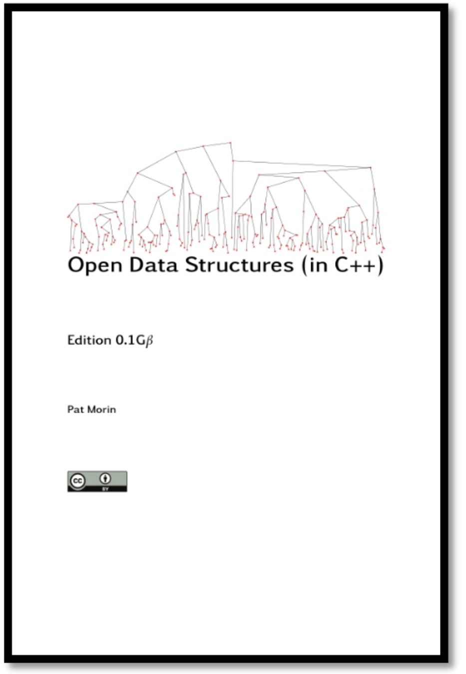
AddFirst(x):
x.next=S.next
S.next.prev=&x
S.next=&x
x.prev=&S

Summary

- Queue ADT: FIFO Queue, Stack (LIFO Queue), Deque
- List ADT: can implement various Queue
- Array based implementations (simple/circular):
  - Queries are fast, updates (i.e., Set) are also fast
  - Modifications (i.e., Add and Remove) are fast at “start” and “end”, but slow in “middle”
  - Capacity can be a problem (come back to this later…)
- Linked list based implementations (singly/doubly linked):
  - Operations (queries, updates, and modifications) are fast at “start” and “end”, but slow in “middle”
  - No capacity issue

Reading

- [CLRS] Ch10 (10.1-10.3)
- [Morin] Ch1 (1.1, 1.2), Ch2 (2.1-2.4), Ch3 (3.1, 3.2)



Application of Stack:

Balancing Symbols

Compiler needs to check whether the parentheses (), brackets [], and braces {} are matched.

<u>CheckParen(str):</u>

```
Stack s
int i=1
while (str[i]!=NULL)
  if (str[i] is ‘(’ or ‘[’ or ‘{’)
    s.push(str[i])
  if (str[i] is ‘)’ or ‘]’ or ‘}’)
    if (s.empty())
      return false
    if (s.pop() and str[i] mismatch)
      return false
  i++
return s.empty()
```

```
if (a>b)
{b=c[10];}
```

```
if (a>b)
{b=c[10];
```

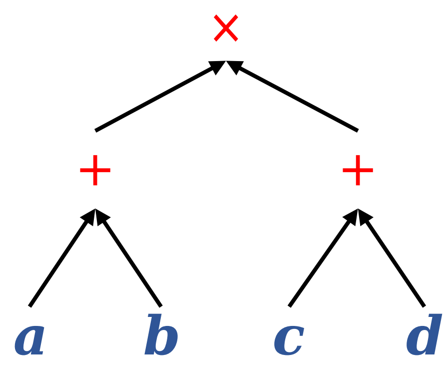
```
if (a>b))
{b=c[10];}
```

```
if (a>b)
{b=c[10);}
```

Application of Stack:
Evaluating Postfix Expressions

How do we evaluate $(a + b) \times (c + d)$?

infix expression



If we place operators **after** operands:

$((a\ b\ +)\ (c\ d\ +)\ \times)$

In fact, we can remove the parentheses:

$a\ b + c\ d + \ \times$

postfix expression

Postfix notation, also known as **reverse Polish notation** (**RPN**), is a mathematical notation in which operators follow their operands. If there are multiple operations, operators are given immediately after their last operands.

RPN does not need parentheses!

One more example:

Infix: $(a + b) \times c + d$

RPN: $a\ b + c \times d\ +$

Application of Stack:
Evaluating Postfix Expressions

Given an expression in RPN, how to evaluate its value?

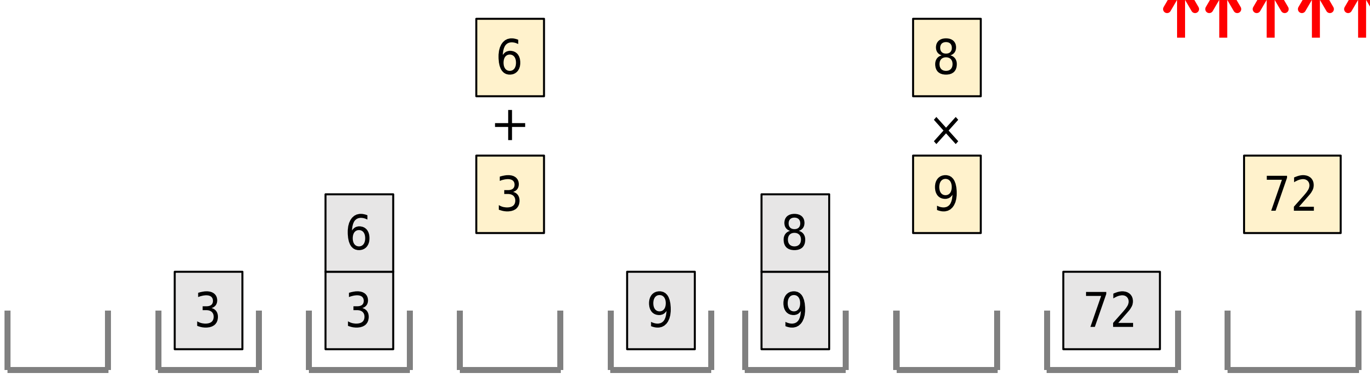
EvalRPN(str):

```
Stack s
while ((token=NextToken(str))!=NULL)
  if (token is an operand)
    s.push(token)
  else
    res=PopOperandAndCalc(s,token)
      s.push(res)
return s.pop()
```

Given an infix expression, how to convert it to RPN and evaluate its value?
(Beware of priorities!)

One simple example:

$3\ 6 + 8 \times$



Application of Stack:

Function Calls

How do function calls actually work?

Alice: only knows addition.

Bob: only knows multiplication.

Question: $100 + 234 + 35 \times 45 + 25$

Calc: 35×45
Answer: $\underline{1575}$
I left at end of line
two

$100 + 234 = 334$
$35 \times$
$334 +$
1909

$35 \times 45 = 1575$

Application of Stack: Function Calls

Call Stack

How do function calls actually work?

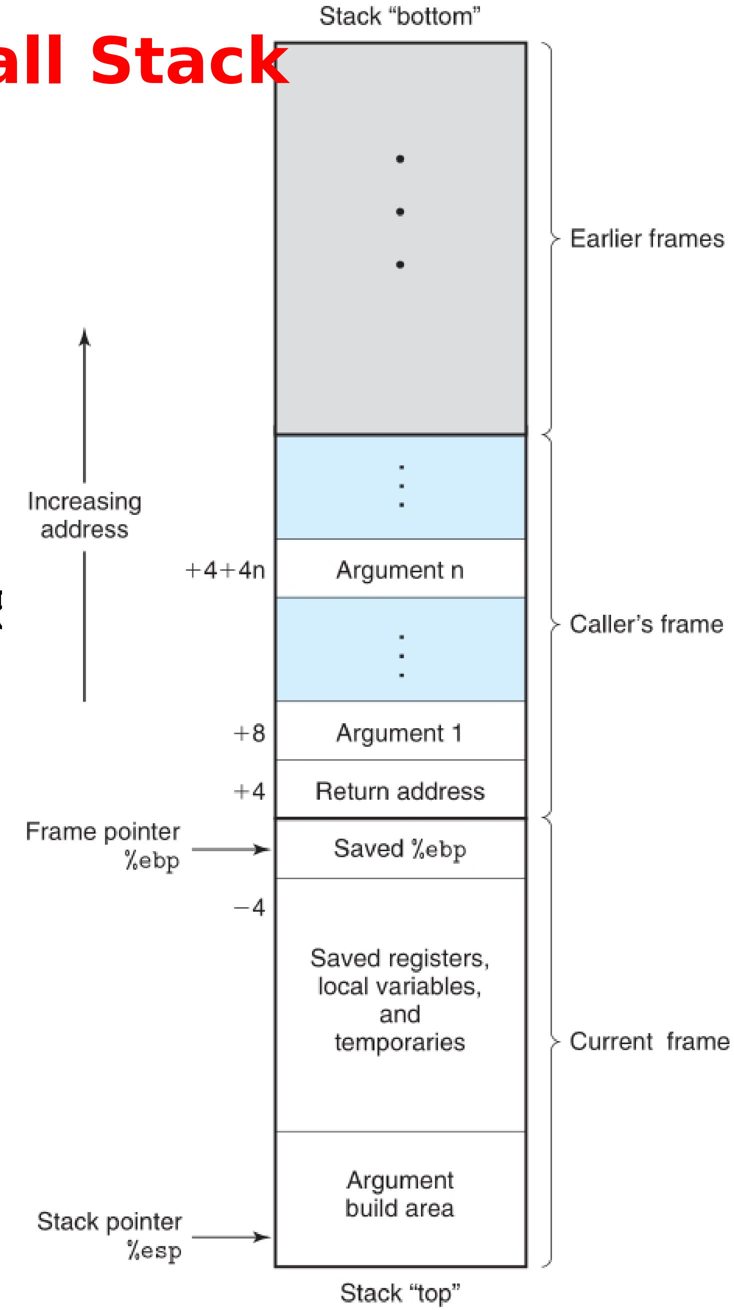
Alice: only knows addition.

Bob: only knows multiplication.

Question: $100 + 234 + 35 \times 45 + 2$

<u>FuncAlice():</u>
sum=100+234
temp=FuncBob(35,45)
sum+=temp
sum+=25
return sum

<u>FuncBob(a,b):</u>
c=a*b
return c



Eliminating Recursion

function calls are implemented via a “call stack”

recursion is a specific type of function call

with the help of a stack, recursion can be replaced by **iteration**

```
FactRec(n):
if (n==1)
  return 1
else
  return n*FactRec(n-1)
```

```
FactIter(n):
Stack s
s.push(Frame(n,-1,NULL))
while (!s.empty())
  frame=s.peek()
  if (frame.val<=1)
    frame.acc=1
  if (frame.acc!=-1) {
    res=(frame.val)*(frame.acc)
    (frame.prevFrame)->acc=res
    s.pop() }
  else
    s.push(Frame(frame.val-1,-1,&frame))
return res
```

“return address” imp

```
struct Frame {
  int val
  int acc
  Frame* prevFrame
}
```

Eliminating Recursion

function calls are implemented via a “call stack”

recursion is a specific type of function call

with the help of a stack, recursion can be replaced by **iteration**

Q: Why recursion can be *undesirable*?
A: Recursion can be slow and memory consuming due to the creation and maintenance of stack frames.

Q: Why recursion can be *desirable*?
A: Recursion can make the code clearer, concise, and intuitive.

Tail Recursion

A function is called **tail-recursive** if each activation of the function will make at most a single recursive call, and will return immediately after that call.

<u>FactRec(n):</u> X

```
if (n==1)
  return 1
else
  return n*FactRec(n-1)
```

<u>EuclidGCDRec(m, n):</u> √

```
if (n==0)
  return m
else
  rem=m%n
  return EuclidGCDRec(n, rem)
```

Tail Recursion

A function is called **tail-recursive** if each activation of the function will make at most a single recursive call, and will return immediately after that call.

Euclid (m, n):

```
if (n==0)
  return m
else
  rem=m%n
  return Euclid(n, rem)
```

Euclid(6, 4):
4==0 ?
rem = 6%4
… … …
return 2

Euclid(4, 2):
2==0 ?
rem = 4%2
… … …
return 2

Euclid(2, 0):
0==0 ?
return 2

Once reaching the base case,
can safely return result **immediately**!

Tail Recursion to Iteration

- Each function parameter is a variable.
- Convert the main body of the function into a loop:
  - *Base cases*: do computation and return results.
  - *Recursive cases*: do computation and update variables.

EuclidGCDRec (m, n):

```
if (n==0)
  return m
else
  rem=m%n
  return EuclidGCDRec(n, rem)
```

EuclidGCDIter (m, n):

```
while (true)
  if (n==0)
    return m
  else
    rem=m%n
    m=n
    n=rem
```

Iteration versus Recursion

- Recursion can be converted into iteration
  - Generic method: simulate a call stack
  - Special case: tail recursion
- Iteration can be converted into tail recursion
- No one is always perfect
  - Iteration can be faster and more memory efficient
  - Recursion can be clearer, more concise and intuitive

Reading

- [Deng] Ch1 (1.4*), Ch4 (4.1-4.4)
- [Weiss] Ch3 (3.6)
- [CSAPP] Ch3 (3.7*)