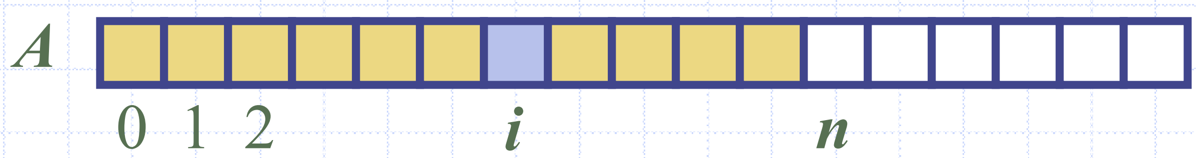
Arrays and Lists

Arrays: Contiguous memory locations => lists

Forms basis for all other data structures

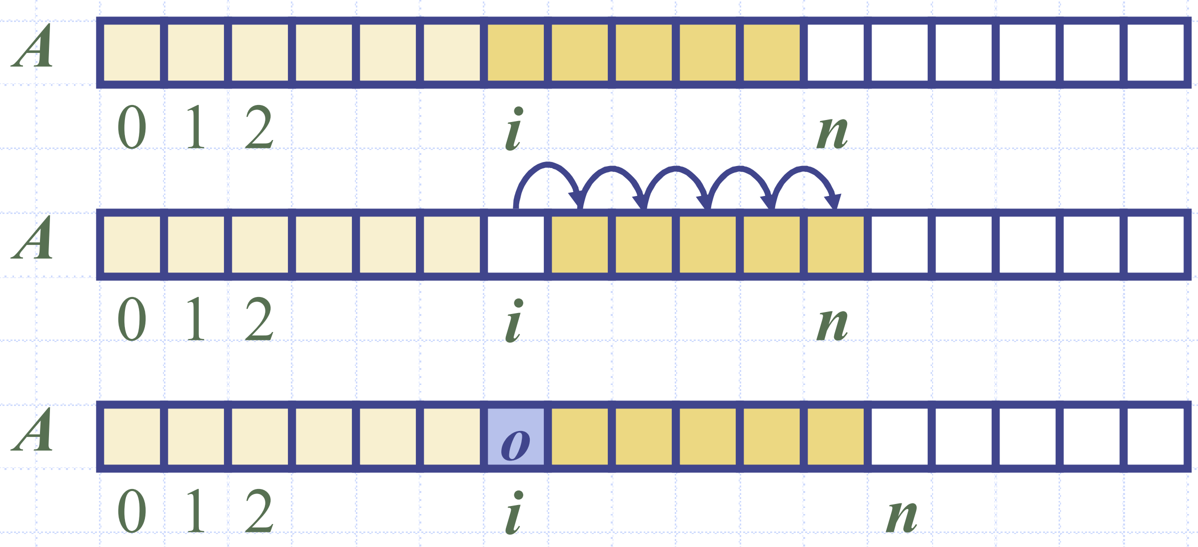
Array-based Implementation

- Use an array A of size N
- A variable n keeps track of the size of the array list (number of elements stored)
- Operation ***at***(i) is implemented in $O(1)$ time by returning $A[i]$
- Operation ***set***(i,o) is implemented in $O(1)$ time by performing $A[i] = o$



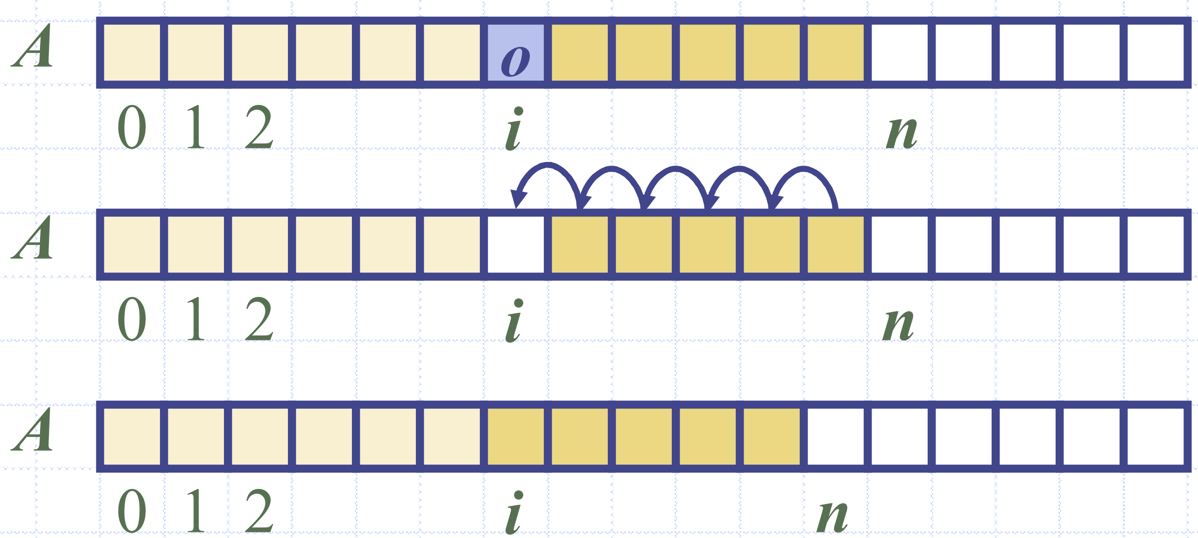
Insertion

- In operation ***insert***(i, o), we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \ldots, A[n-1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Element Removal

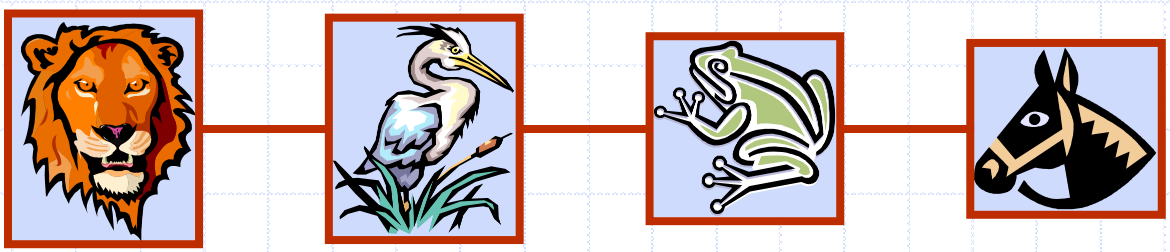
- In operation *erase*(i), we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i + 1], \ldots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Performance

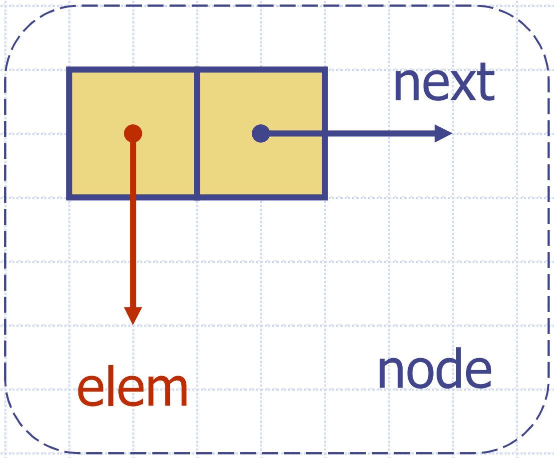
- In the array based implementation of an list:
  - The space used by the data structure is $O(n)$
  - ***size***, ***empty***, ***at*** and ***set*** run in $O(1)$ time
  - ***insert*** and ***erase*** run in $O(n)$ time in worst case
- If we use the array in a circular fashion, operations ***insert***$(0, x)$ and ***erase***$(0, x)$ run in $O(1)$ time
- In an ***insert*** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

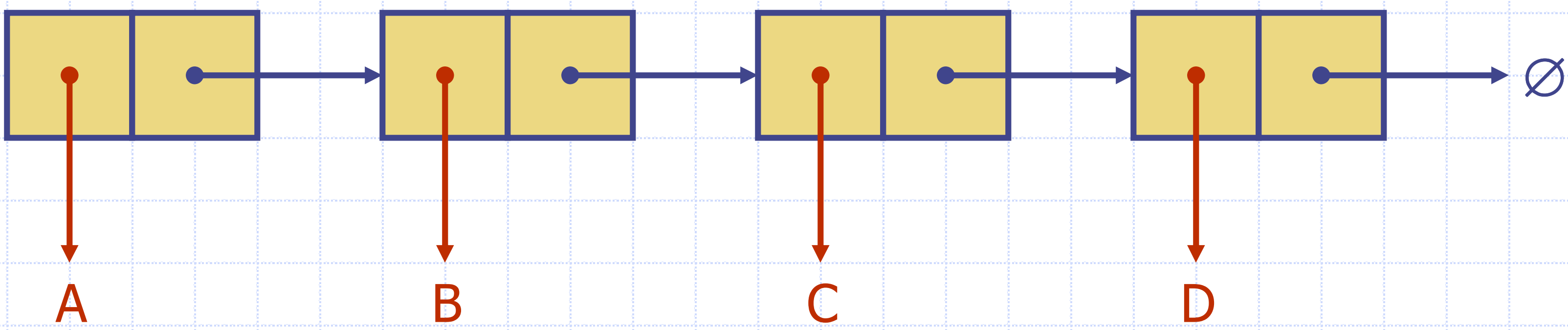
Linked Lists



Singly Linked List (§ 3.2)

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node





```
insertBefore( Node B, int v ) {
    Node tmp = head;
    while ( tmp.next != b ) {
        tmp = tmp.next;
    }
    nd.val = v;
    nd.next = b;
    tmp.next = nd;
}
```

```
Delete( int v ) {
    tmp = head;
    if ( tmp.val == v ) {
        head = tmp.next;
    }
    while ( tmp.next.val != v ) {
        tmp = tmp.next;
    }
    tmp.next = tmp.next.next;
}
```

- Write a recursive algorithm to add all values of the nodes in a single linked list.
- Main() {
-     s = sum(head);
- }
- Int sum(NODE tmp) {
-     if (tmp == null) { return 0 }
-      else return tmp.data + sum(tmp.next)

- Write re cursive function, to find the largest value in a single linked list. There is ATLEAST one node. DO NOT ASSUME any value.
- Main() {
-     v = large(head);
- }
- Int large(tmp) {
-    if (tmp.next == null) { return tmp.val }
-        v = large(tmp.next);
-         if (v > tmp.val) return v; else return tmp.val;

- Write recursive algorithm to count the number of nodes in the single linked list. If the list is empty, return 0
- Main() {
-     n = count(head);
- }
- Int count(tmp) {
-       if (tmp == null) { return 0 }
-       else return count(tmp.next) + 1;

- Void Delete(Node tmp, Node *n) {
  if (tmp->next == n) {
-     tmp.next = n.next;
- }
- Else delete(Tmp->next, n);

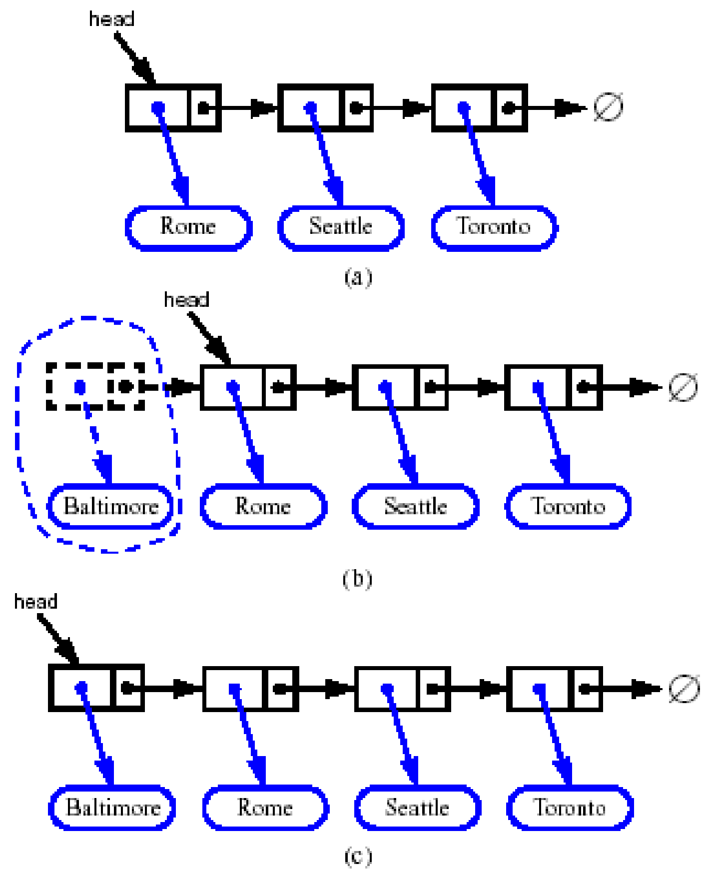
- Write recursive algorithm to count the number of nodes with value v. Could be 0 nodes, or more than 1 node.
- Main() {
-     n = countv(head,v);
- }
- Int countv(tmp,v) {
-   if (tmp == null) return 0;
-    else if (tmp.val == v) return 1 + countv(tmp.next,v);
-    else return countv(tmp.next,v)

The single linked list has several nodes with value v. Count the nodes that have the same value v.

```
Int Count( Node tmp, v ) {
        if ( tmp == null ) { return 0; }
        else if ( tmp.val == v ) { return 1 + count(
tmp->next, v ); }
         else { return 0+count(tmp.next, v) };
}
```

Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node



- insertAtEnd(int v) {

- Node *nd;
- Node *tmp;
- tmp = head;
- while (tmp->next != null) {
- tmp = tmp->next;
- }
- nd.val = v;
- tmp->next = nd;
- nd->next = null;
-

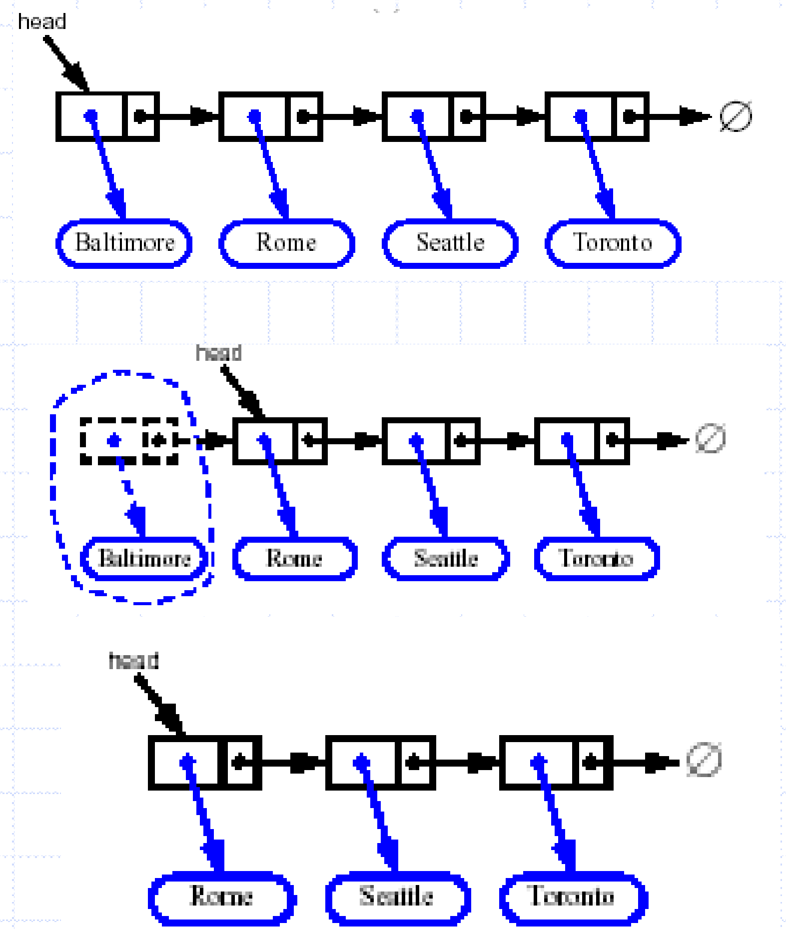
- Main() {
- insertAtEnd(head, v);
- }
- Void insertAtEnd(tmp, v) {
- if (tmp->next == null) {
- nd = new ….
- nd.val = v;
- nd.next = null;
- tmp.next = nd;
- }
- insertAtEnd(tmp->next, v);
- }

- insertAfter(Node *b, int v) {
-

```
Main() {
      insertAtEnd( head, v );
}
Void insertAtEnd( tmp, v ) {
    if (tmp->next == null ) {
        nd = new ….
         nd.val = v;
         nd.next = null;
         tmp.next = nd;
    }
    insertAtEnd(tmp->next, v );
}
```

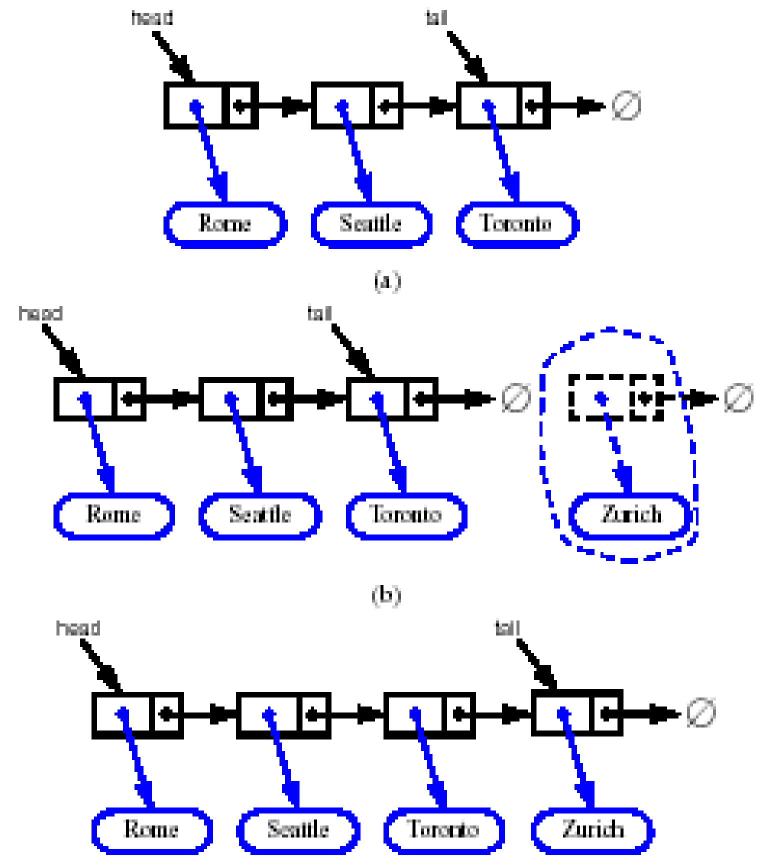
Removing at the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node



Inserting at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node



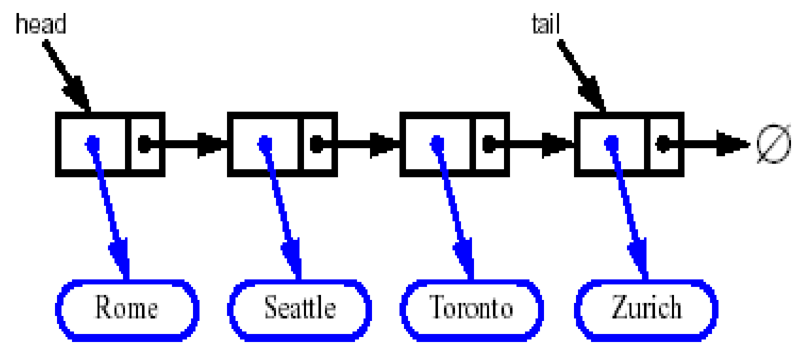
(a)

(b)

(c)

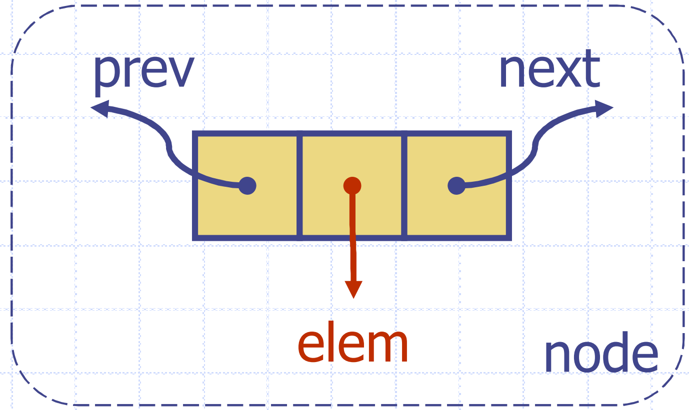
Removing at the Tail

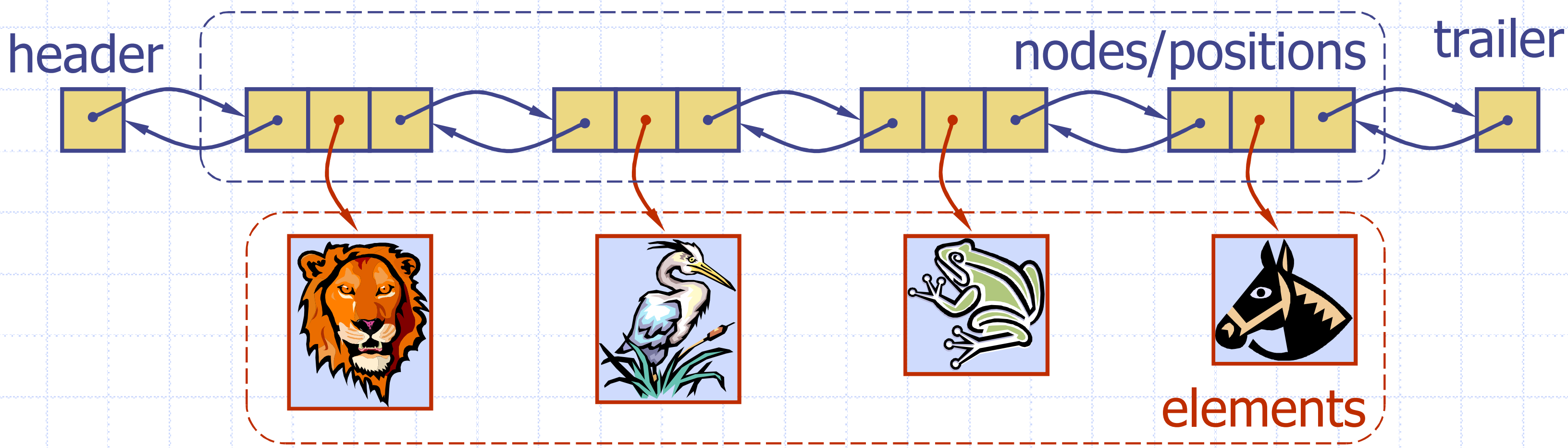
- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node



Doubly Linked List

- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





- insertAfter(node b, int v) {
- node nd;
- nd.val = v;
- nd.next = b.next;
- nd.prev = b;
- c = b.next; c.prev = nd
- b.next.prev = nd;
- b.next = nd;
-

- Delete(n) {
-     if (n == head) {
-         head = head.next;
-          if (head == null) { tail = null }
-          else head.prev = null;
-     }
-     else   if (n == tail) {   tail = tail.prev; tail.next = null; }
-      else { n.prev.next = n.next; n.next.prev = n.prev; }
-

- insertBefore(node b, int v){
- nd.val = v;
- nd.next = b;
- nd.prev = b.prev.
- b.prev.next = nd;
- b.prev = nd;

- //Given single linked list, contains at least one node, return sum of the values in the list nodes
- Int sum(head) {
- if (head == null) {
- return 0;
- }
- else {return (head.val + sum(head.next) }
- }
- //Given single linked list, contains at least one node, return greatest value node in the list
- Greatest(head) {
- if (head.next == null) {
- return head.val;
- }
- else {
- tmp = greatest(head.next);
- if (head.val > tmp) { return head.val }
- else { return tmp; }
- }

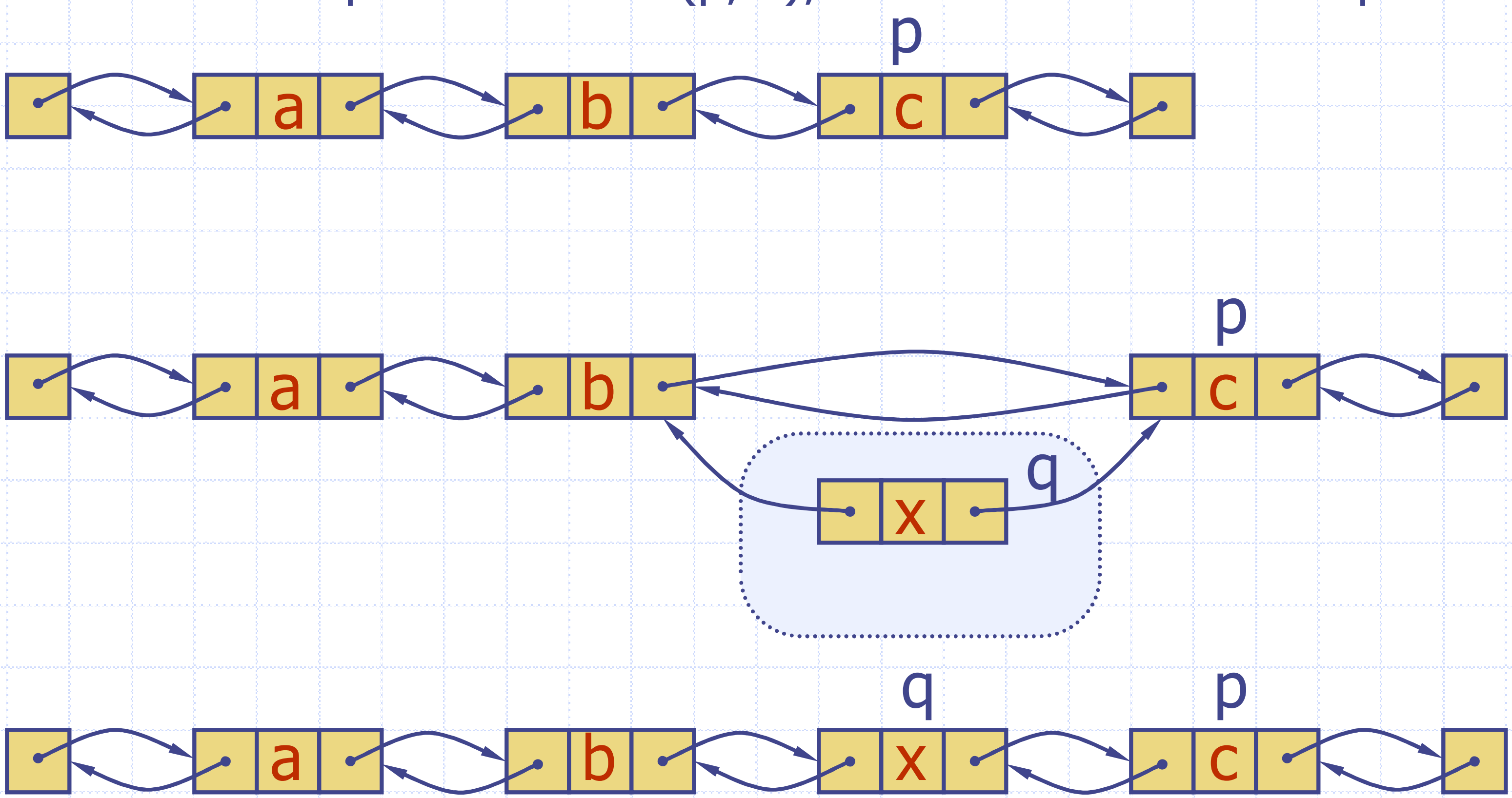
```
Node {
        int key;
        Node *next;
        Node *prev;
}
Rev_trav(tail) {
        if ( tail == null )  {  return; }
        print tail.key
        rev_trav(tail.prev);
}
```

- Given double linked list. Assume the number of nodes is odd. You have head and tail. Write a recursive algorithm to find the middle.
- Main() {
-     node = mess(head,head.next);
- }
- bool mess(tmp1, tmp2) {
-     if (tmp1 == tmp2) { return true; }
-     if (tmp2 == null) return false;
-     if (tmp2.next == null) { return false;
-     return mess(tmp1.next, tmp2.next.next)
- }
-

- Given - single linked list, head
- At least 3 nodes
- Odd number of nodes in the list
- Find middle node
- Recursive function
- Main ( ) { middle(head, head);
- Middle(head, n) {
-       if (n.next == null) { return head }
-       else { return middle(head.next, n.next.next) ;

Insertion

- We visualize operation insert(p, x), which inserts x before p



Insertion Algorithm

Algorithm insert(p, e): {insert e before p}

Create a new node v

v→element = e

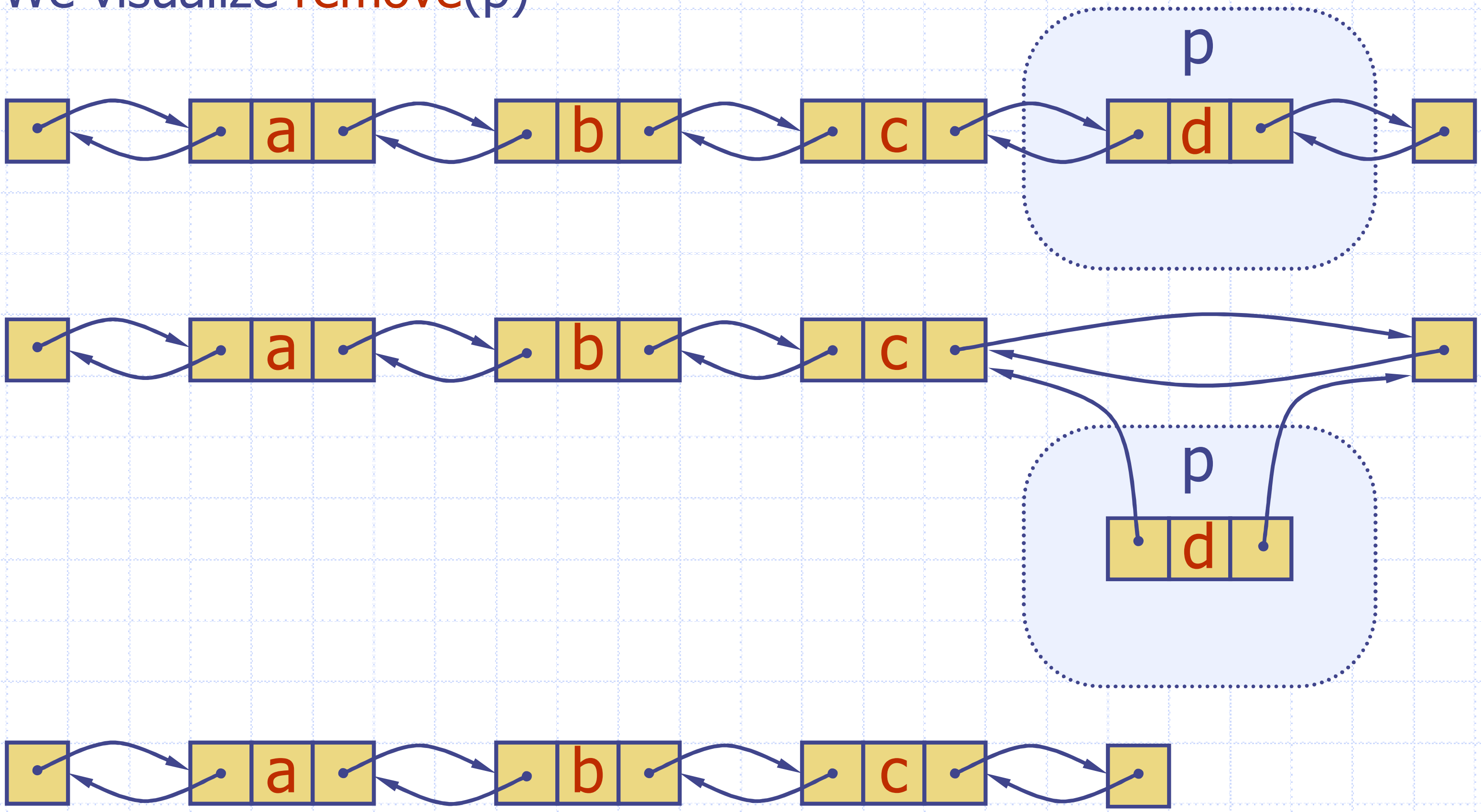
u = p→prev

v→next = p; p→prev = v {link in v before p}

v→prev = u; u→next = v {link in v after u}

Deletion

- We visualize remove(p)



- Single linked list. Write recursive function that returns the largest node in the list.

```
Largest(head) {
        if ( head == null ) {
              return invalid;
        }
        if ( head->next == null )  {
                return( head.val );
        }
        else return( max( head.val, largest(head.next) );
 }
```

Deletion Algorithm

Algorithm remove(p):

u = p→prev

w = p→next

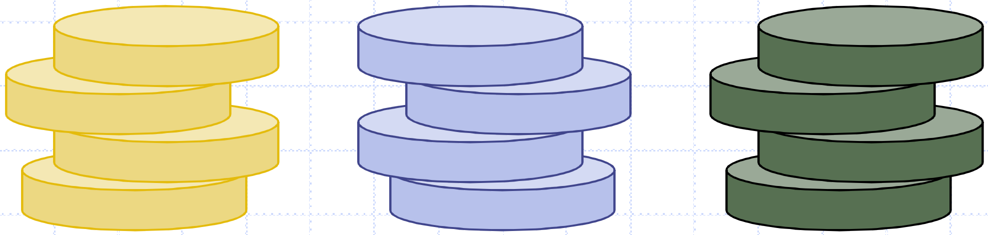
u→next = w {linking out p}

w→prev = u

Performance

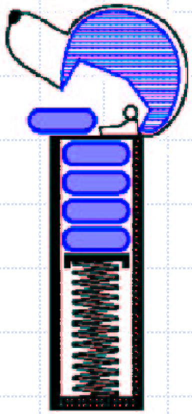
- In the implementation of the List ADT by means of a doubly linked list
  - The space used by a list with n elements is $O(n)$
  - The space used by each position of the list is $O(1)$
  - All the operations of the List ADT run in $O(1)$ time
  - Operation element() of the Position ADT runs in $O(1)$ time

Stacks



The Stack ADT

- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes the last inserted element



- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean empty(): indicates whether no elements are stored

Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be "thrown" by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting pop or top on an empty stack throws a StackEmpty exception

Applications of Stacks

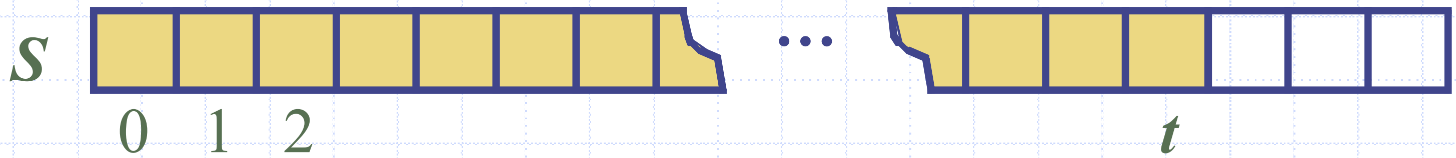
- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the C++ run-time system
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

Algorithm *size*()
  return $t + 1$

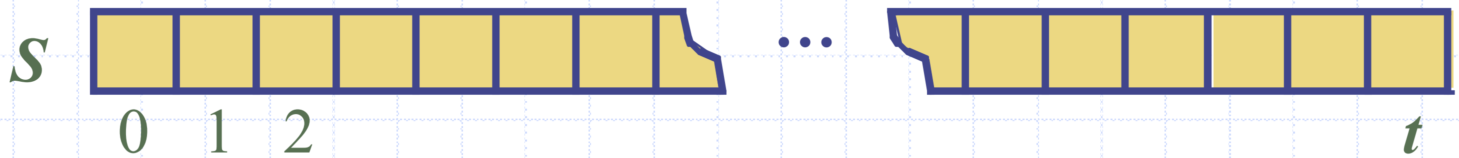
Algorithm *pop*()
  if *empty*() **then**
    throw *StackEmpty*
  else
    $t \leftarrow t - 1$
    return $S[t + 1]$



Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a StackFull exception
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

Algorithm $push(o)$
  if $t = S.size() - 1$ **then**
    throw *StackFull*
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$



Performance and Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined a priori and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

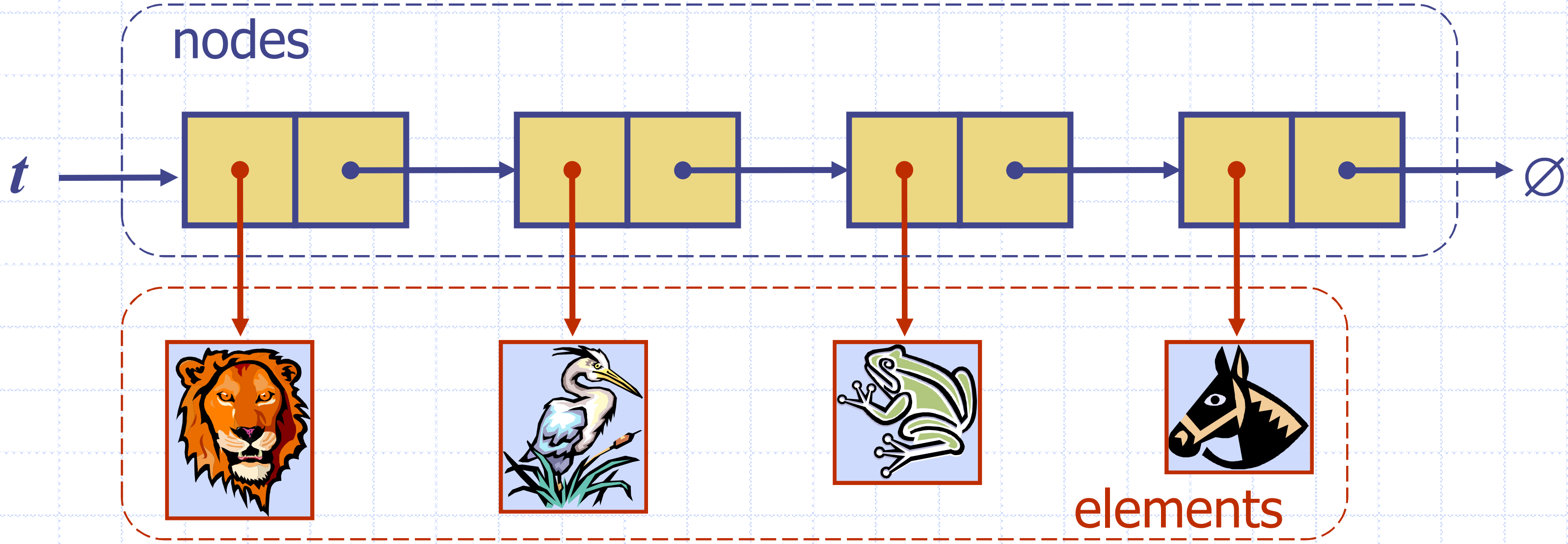
Array-based Stack in C++

```
template <typename E>
class ArrayStack {
private:
    E* S; // array holding the stack
    int cap; // capacity
    int t; // index of top element
public:
    // constructor given capacity
    ArrayStack( int c) :
        S(new E[c]), cap(c), t(-1) { }
```

```
    void pop() {
      if (empty()) throw StackEmpty
              ("Pop from empty stack");
        t--;
    }
    void push(const E& e) {
        if (size() == cap) throw
            StackFull("Push to full stack");
        S[++ t] = e;
}
… (other methods of Stack interface)
```

Stack as a Linked List (§ 5.1.3)

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time



Example use in C++

* indicates top

```
ArrayStack<int> A;                          // A = [ ], size = 0
A.push(7);                                  // A = [7*], size = 1
A.push(13);                                 // A = [7, 13*], size = 2
cout << A.top() << endl; A.pop();           // A = [7*], outputs: 13
A.push(9);                                  // A = [7, 9*], size = 2
cout << A.top() << endl;                    // A = [7, 9*], outputs: 9
cout << A.top() << endl; A.pop();           // A = [7*], outputs: 9
ArrayStack<string> B(10);                   // B = [ ], size = 0
B.push("Bob");                              // B = [Bob*], size = 1
B.push("Alice");                            // B = [Bob, Alice*], size = 2
cout << B.top() << endl; B.pop();           // B = [Bob*], outputs: Alice
B.push("Eve");                              // B = [Bob, Eve*], size = 2
```

Parentheses Matching

- Each "(", "{", or "[" must be paired with a matching ")", "}", or "["
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Parentheses Matching Algorithm

Algorithm ParenMatch(*X*,*n*):
Input: An array *X* of *n* tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number
Output: **true** if and only if all the grouping symbols in *X* match
Let *S* be an empty stack
for *i*=0 to *n*-1 **do**
  if *X*[*i*] is an opening grouping symbol **then**
    S.push(*X*[*i*])
  else if *X*[*i*] is a closing grouping symbol **then**
    if *S*.empty() **then**
      return false {nothing to match with}
    if *S*.pop() does not match the type of *X*[*i*] **then**
      return false {wrong type}
if *S*.empty() **then**
  return true {every symbol matched}
else return false {some symbols were never matched}

Evaluating Arithmetic Expressions

Slide by Matt Stallmann included with permission.

14 – 3 * 2 + 7 = (14 – (3 * 2)) + 7

Operator precedence

* has precedence over +/–

Associativity

operators of the same precedence group
evaluated from left to right
Example: (x – y) + z rather than x – (y + z)

Idea: push each operator on the stack, but first pop and perform higher and *equal* precedence operations.

Algorithm for Evaluating Expressions

Slide by Matt Stallmann included with permission.

Two stacks:

- opStk holds operators
- valStk holds values
- Use $ as special "end of input" token with lowest precedence

Algorithm doOp()

```
x ← valStk.pop();
y ← valStk.pop();
op ← opStk.pop();
valStk.push( y op x )
```

Algorithm repeatOps(refOp):

```
while ( valStk.size() > 1 ∧
        prec(refOp) ≤
        prec(opStk.top())
   doOp()
```

Algorithm EvalExp()

Input: a stream of tokens representing an arithmetic expression (with numbers)

Output: the value of the expression

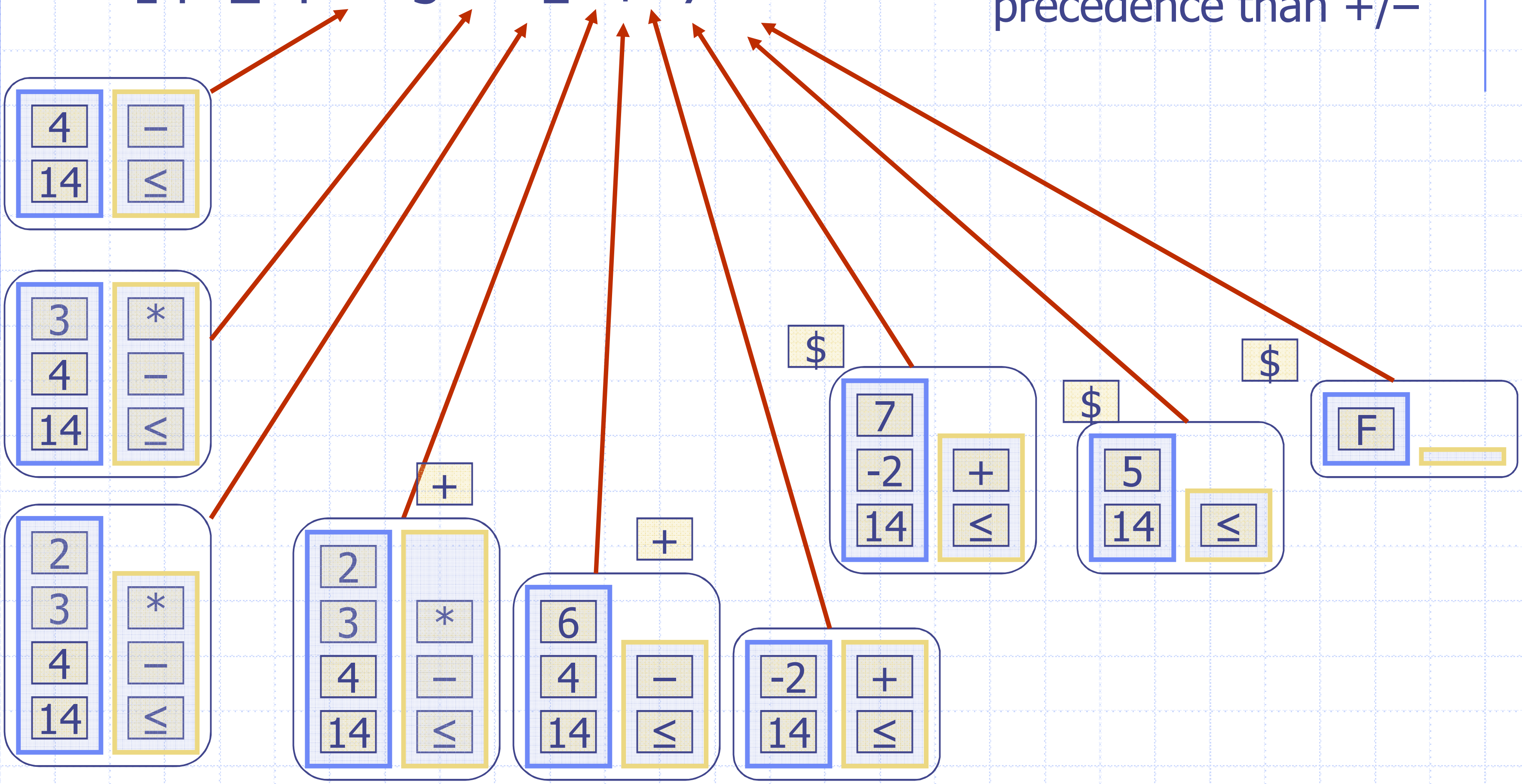
```
while there's another token z
   if isNumber(z) then
        valStk.push(z)
   else
        repeatOps(z);
        opStk.push(z)
repeatOps($);
return valStk.top()
```

Algorithm on an Example Expression

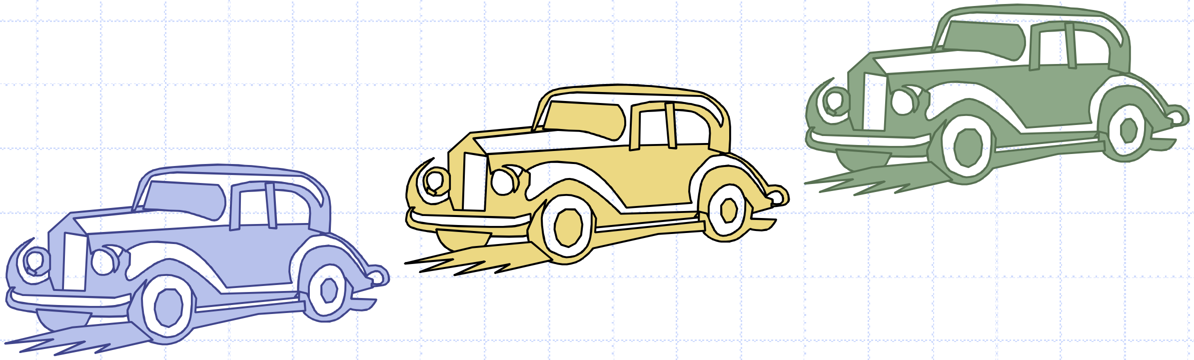
Slide by Matt Stallmann included with permission.

14 ≤ 4 – 3 * 2 + 7

Operator ≤ has lower precedence than +/–



Queues



The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - dequeue(): removes the element at the front of the queue
- Auxiliary queue operations:
  - object front(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean empty(): indicates whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an QueueEmpty

Example

Operation	***Output***	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	5	(3)
enqueue(7)	–	(3, 7)
dequeue()	–	(7)
front()	*7*	(7)
dequeue()	–	()
dequeue()	*“error”*	()
empty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	–	(7, 3, 5)

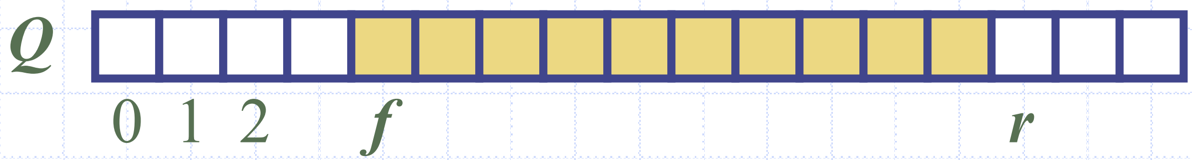
Applications of Queues

- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

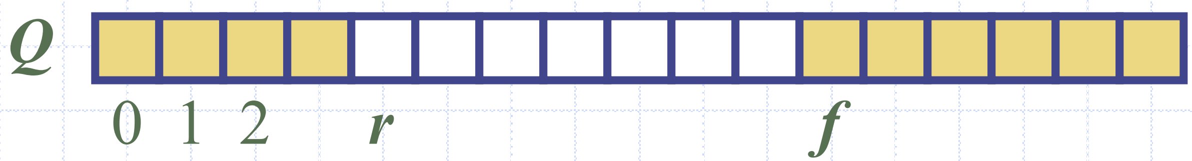
Array-based Queue

- Use an array of size N in a circular fashion
- Three variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
  - n number of items in the queue

normal configuration



wrapped-around configuration

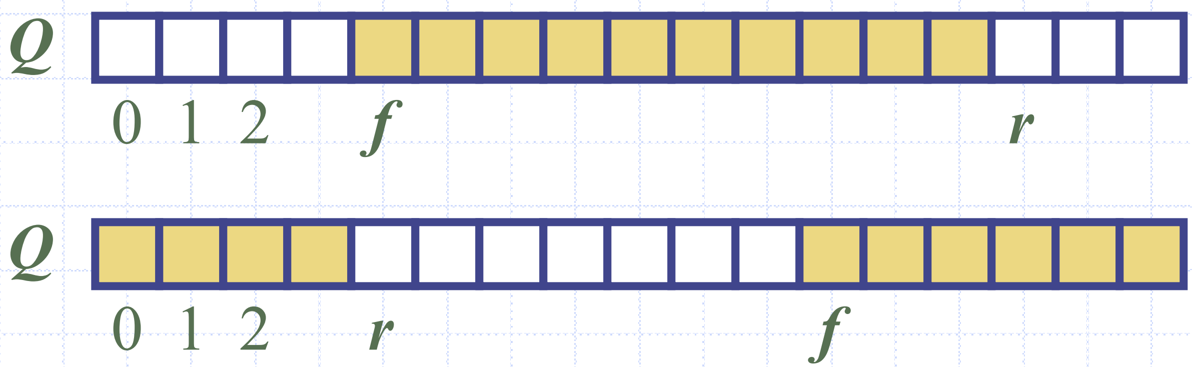


Queue Operations

- Use *n* to determine size and emptiness

Algorithm *size*()
  return *n*

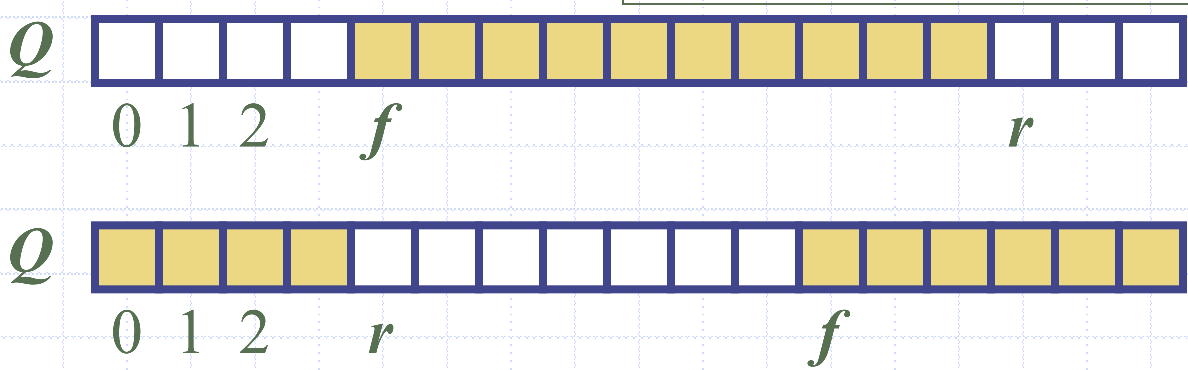
Algorithm *empty*()
  return (*n* = 0)



Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

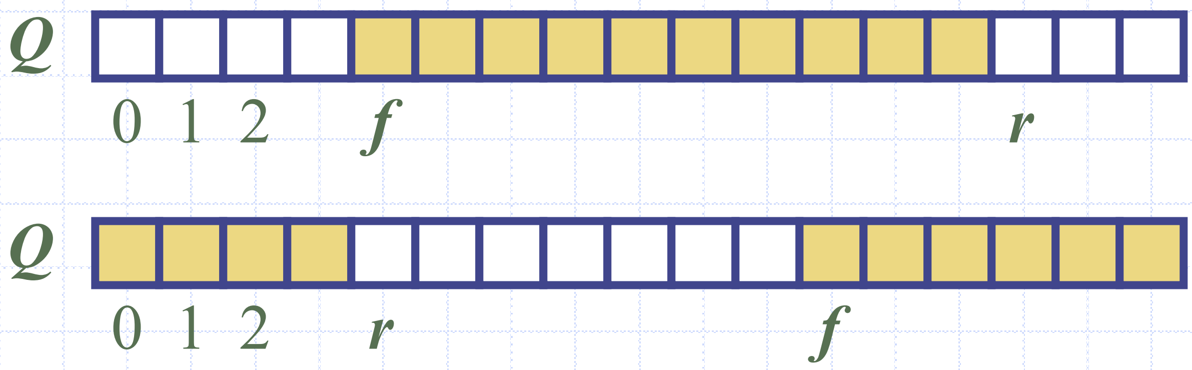
Algorithm *enqueue*(o)
  if *size*() $= N - 1$ **then**
    throw *QueueFull*
  else
    $Q[r] \leftarrow o$
    $r \leftarrow (r + 1) \bmod N$
    $n \leftarrow n + 1$



Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

Algorithm *dequeue*()
  if *empty*() **then**
    throw *QueueEmpty*
  else
    $f \leftarrow (f + 1) \bmod N$
    $n \leftarrow n - 1$



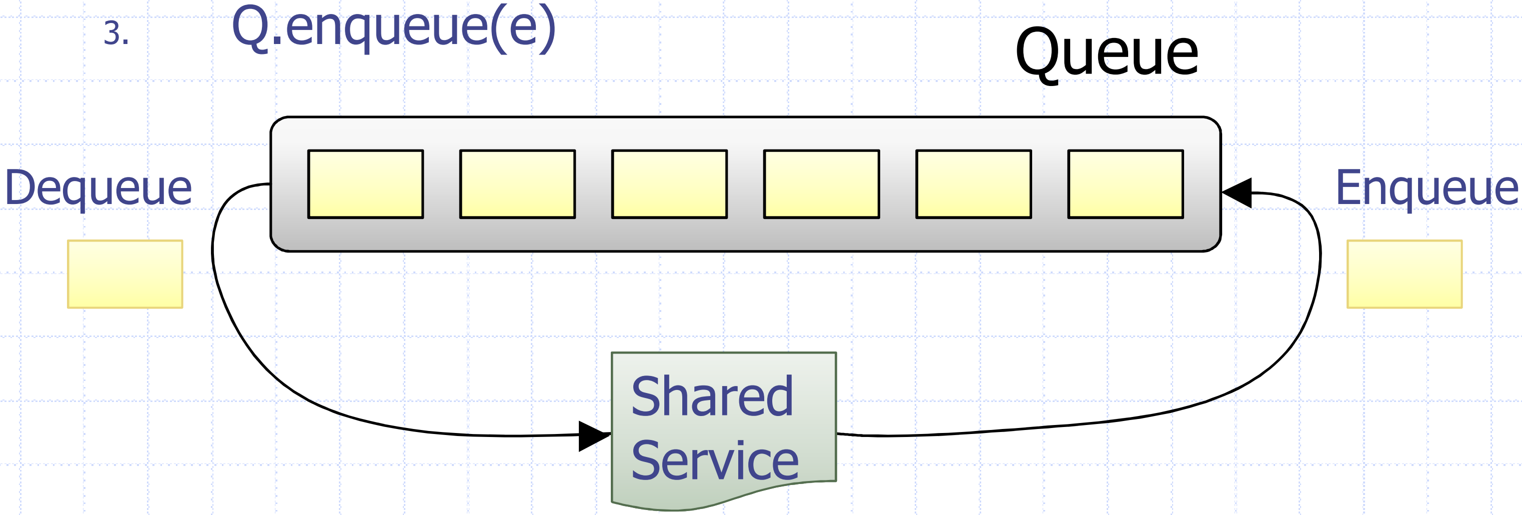
Queue Interface in C++

- C++ interface corresponding to our Queue ADT
- Requires the definition of exception QueueEmpty
- No corresponding built-in C++ class

```
template <typename E>
class Queue {
public:
  int size() const;
  bool empty() const;
  const E& front() const
    throw(QueueEmpty);
  void enqueue (const E& e);
  void dequeue()
    throw(QueueEmpty);
};
```

Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:
  1. e = Q.front(); Q.dequeue()
  2. Service element e
  3. Q.enqueue(e)



Queue as a Linked List

- We can implement a queue with a singly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time