

ADT: Lists, Stacks and Queues

N:6, 7, 8, 11

Outline

▶ List as an ADT

- ▶ An array-based implementation of lists
- ▶ Linked lists with pointer implementation

▶ Stacks

- ▶ Operations and implementations
- ▶ Applications: decimal to binary conversion, parenthesis matching, infix to postfix, postfix computation, expression tree, etc.

▶ Queues

- ▶ Operations and implementations
- ▶ Bin-sort and radix sort

Consider Every Day Lists

- ▶ Groceries to be purchased
- ▶ Job to-do list
- ▶ List of assignments for a course
- ▶ Dean's list

- ▶ Can you name some others??



Properties of Lists

- ▶ Can have a single element
- ▶ Can have no elements
- ▶ There can be lists of lists

- ▶ We will look at the list as an abstract data type
 - ▶ Homogeneous data type (i.e., no mixed types on int, double, string, etc.)
 - ▶ Sequential elements (elements are indexed from 0 to size-1)

Basic Operations

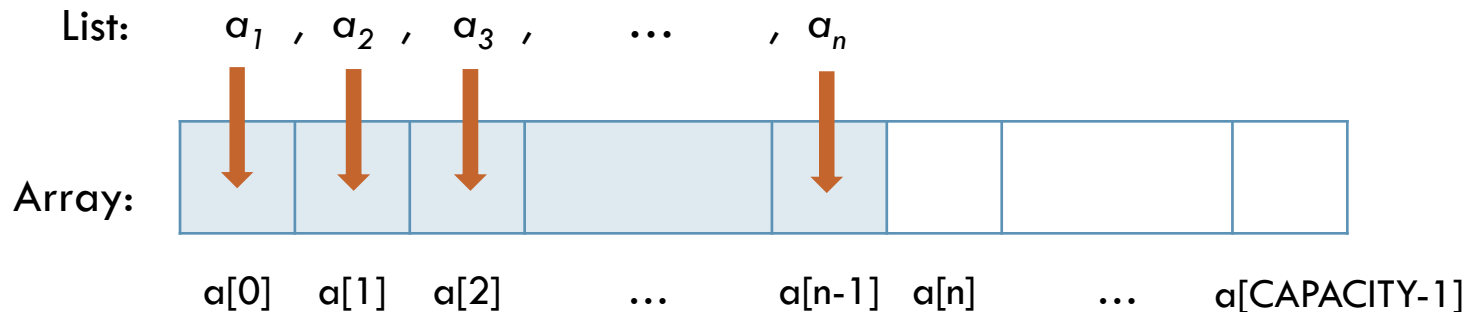
- ▶ Construct an empty list
- ▶ Determine whether or not empty
- ▶ Insert an element into the list
- ▶ Delete an element from the list
- ▶ Traverse (iterate through) the list to
 - ▶ Modify
 - ▶ Output
 - ▶ Search for a specific value
 - ▶ Copy or save
 - ▶ Rearrange

Designing a **List** Class

- ▶ Should contain at least the following function members
 - ▶ Constructor
 - ▶ `empty()`: whether the list is empty or not
 - ▶ `insert()`: insert an element into the list
 - ▶ `delete()`: delete an element from the list
 - ▶ `display()`: display all the elements in the list
- ▶ Implementation involves
 - ▶ Defining data members
 - ▶ Defining function members from design phase

Array-Based Implementation of Lists

- ▶ An array is a viable choice for storing list elements
 - ▶ Element are sequential
 - ▶ It is a commonly available data type
 - ▶ Algorithm development is easy
- ▶ Normally sequential orderings of list elements match with array elements



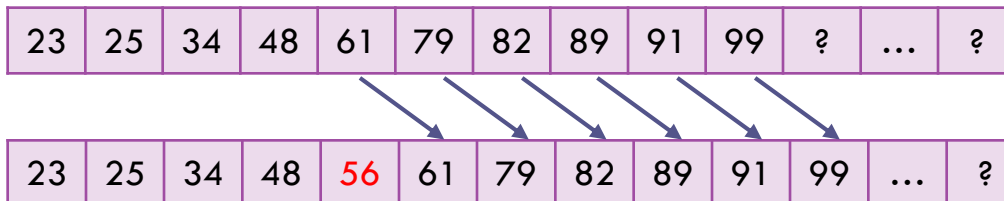
Implementing Operations

- ▶ **Constructor**
 - ▶ Static array allocated at compile time
- ▶ **Empty**
 - ▶ Check if `size == 0`
- ▶ **Traverse**
 - ▶ Use a loop from 0th element to `size - 1`

Implementing Operations

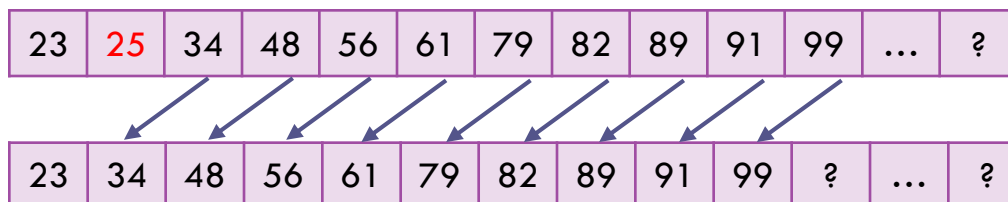
▶ Insert

- ▶ Shift elements to right of insertion point



▶ Delete

- ▶ Shift elements back



Also adjust **size**
up or down

List Class Example (Array Implementation)

- ▶ Declaration file (**List.h**)
 - ▶ Note use of **typedef** mechanism outside the class
 - ▶ This typedef means that it is a list of **int**
- ▶ Definition and implementation (**List.cpp**)
 - ▶ Note considerable steps required for **insert()** and **erase()** functions
- ▶ Program to test the class (**listtester.cpp**)

List Class with Static Array

- ▶ Must deal with issue of declaration of the type and the size **CAPACITY**

- ▶ Use **typedef** mechanism to define type for the WHOLE program

```
typedef Some_Specific_Type ElementType  
ElementType array[CAPACITY];
```

- ▶ For specific implementation of our class we simply fill in desired type for **Some_Specific_Type**

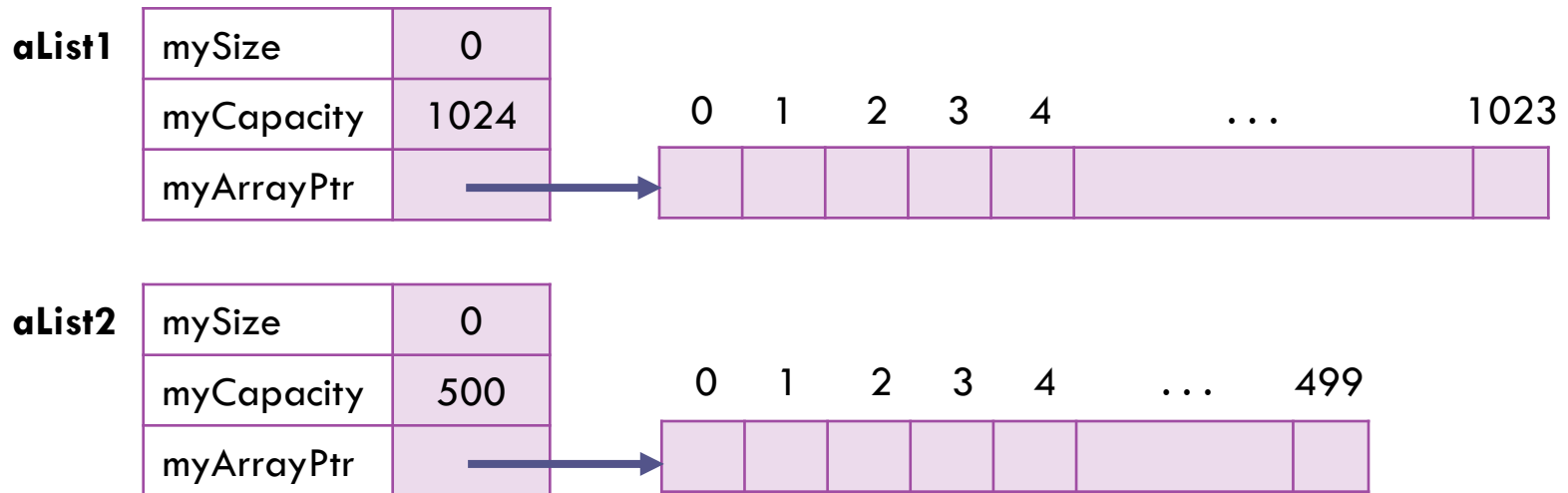
List Class with Static Array: Problems

- ▶ `capacity`: Stuck with "one size fits all"
 - ▶ Too large: Could be wasting space
 - ▶ Too small: Could run out of space
- ▶ Thus we consider creating a List class with dynamically-allocated array
 - ▶ The capacity can be changed at object construction

Dynamic-Allocation for List Class

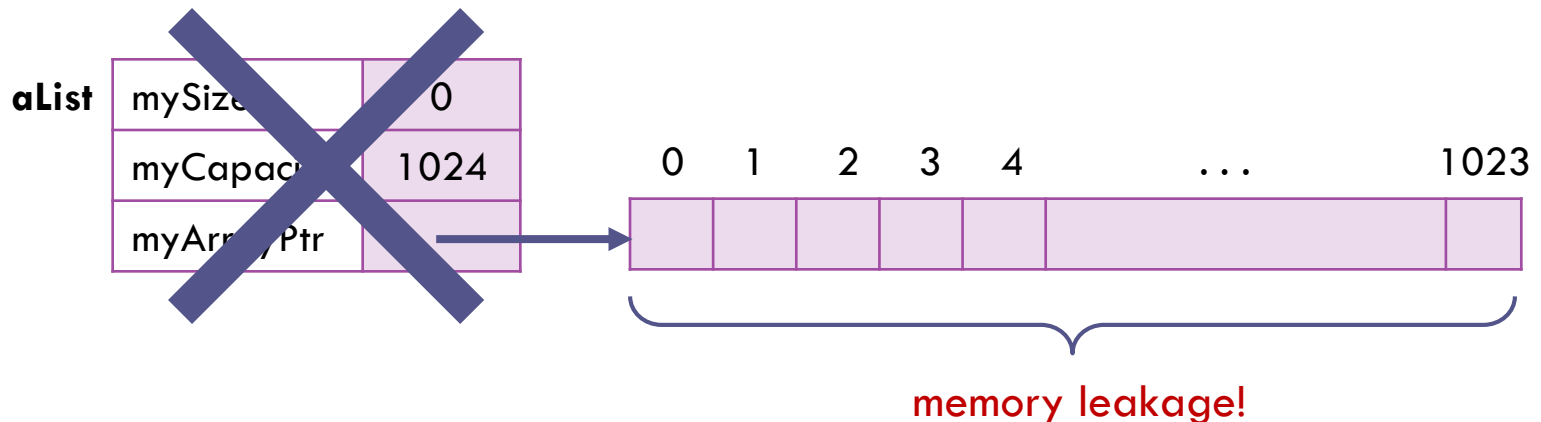
- ▶ Now possible to specify different sized lists

```
cin >> maxSize;  
List aList1, aList3(maxSize);  
List aList2 (500);
```



Need to Implement Destructor

- ▶ When class object goes out of scope the pointer to the dynamically allocated memory is not reclaimed automatically
- ▶ The destructor reclaims dynamically allocated memory



Copy Constructor

- ▶ **Called for construction statement like `foo a=b;`**
 - ▶ a and b are of the *same* class objects foo
 - ▶ Note that this is the same as `foo a(b);` and hence we overload the constructor this way
 - ▶ For the data members not covered in the copy constructor, their copy constructors will be called (primitive type are simply memory copy)
- ▶ **Compiler provides a default copy constructor if you do not have one**
 - ▶ Copy each member of the original object into the corresponding member of the new object (i.e., calling the copy constructor of each data member)
 - ▶ Pointers are not traversed and hence a and b may point to the same heap location
 - ▶ Can cause serious problems when data members contain pointers to dynamically allocated memory → this can cause memory problem when objects are copied into and returned from functions as parameters

The need of deep copy

- ▶ **Need to make a *deep copy* of an object**
 - ▶ When object is initialized by another in a declaration (`List lst(12);`)
 - ▶ When argument is passed as value parameter in functions (`foo(List lst);`)
 - ▶ When function returns a local object (`List lst = bar();`)
 - ▶ When temporary storage of object is needed (`List tmp=aList;`)
- ▶ **We want to make duplications on the heap that an object points to**
 - ▶ Instead of doing shallow copy

Copy Constructor Statement

- ▶ We need to declare copy constructor:

- ▶ As a member function

```
class foo{
public:
    foo( const foo & f ); // must have const
...};
...
foo g;
```

and then call

```
foo f( g );
```

or

```
foo f = g; // foo f = foo(g) and then call it as f
```

- ▶ Note that the statement

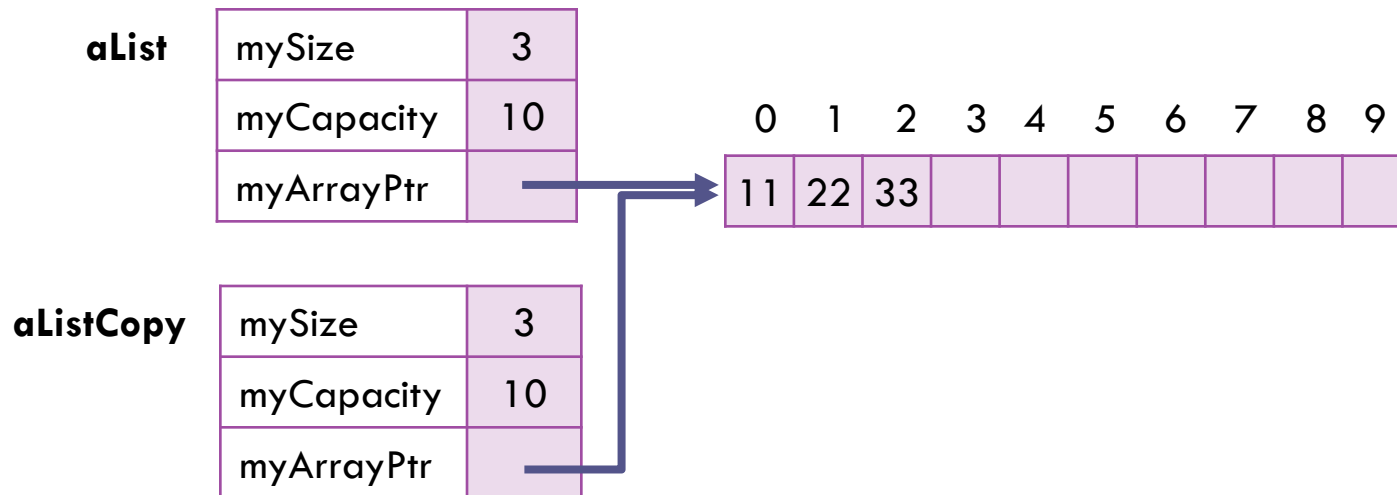
```
int i;
i(2); // NOT a copy constructor
```

is not valid as the compiler will take `i(2)` as a function call instead of setting `i=2`. Use `int i(2);` instead

Copy Constructor with Deep Copy

- ▶ Deep copy: enables pass-by-value and return-by-value between objects
 - ▶ Used to copy original object's values into new object to be passed to a function or returned from a function
- ▶ If copy is not made deep (shallow copy), aliasing problem

```
List aListCopy( aList );
```



Conversion Constructor

- ▶ The statement `foo f = bar` or `foo f (bar)`, where `bar` is of `bar_class` class (NOT of `foo` class), constructs an object by *converting* an object of a *different* class
- ▶ It calls the conversion constructor
- ▶ You need to define a conversion constructor to tell compiler how to deal with the above, using

- ▶ `foo(bar_class b);`

- ▶ For example, you can convert a real number to a complex number by statement like:

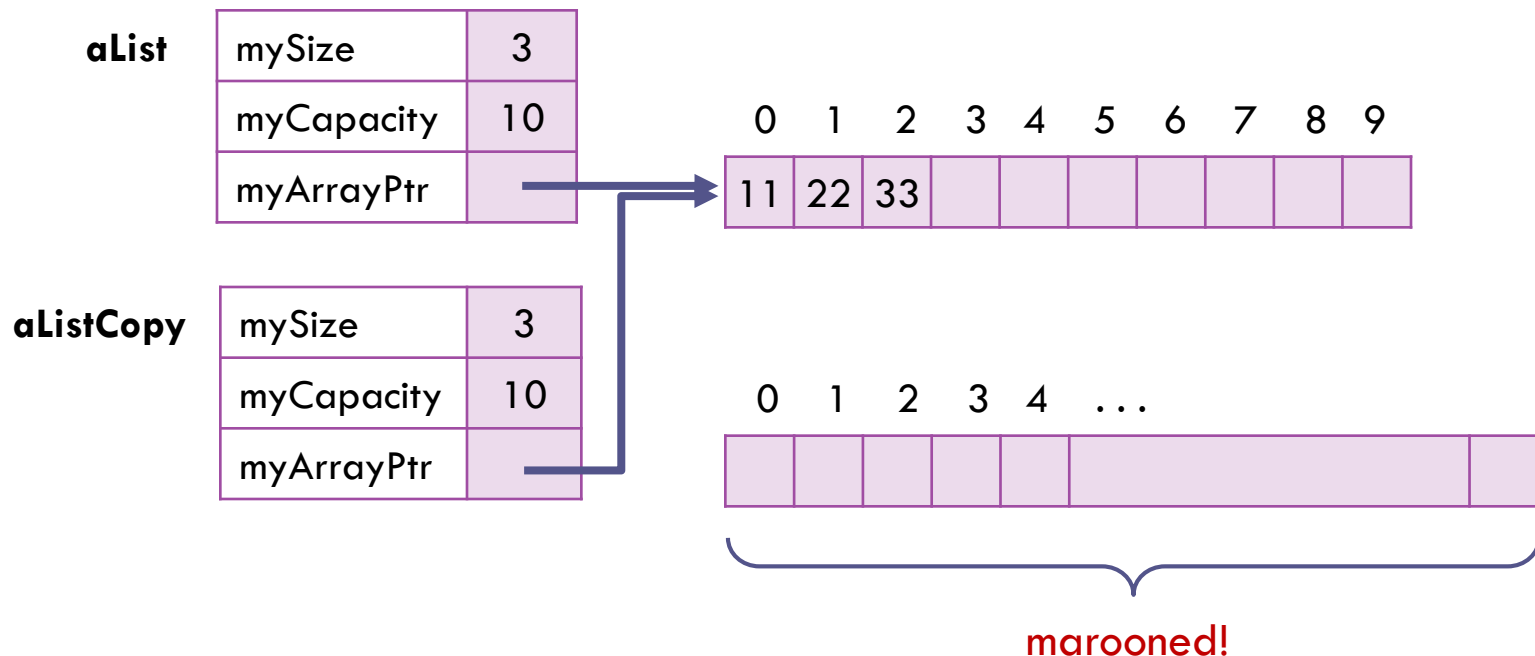
`Complex c(0.2)` or `Complex c = 0.2;`

You need to define `Complex(double d);` in your class definition.

Assignment Operator

- ▶ Default assignment operator makes shallow copy
- ▶ Can cause memory leak if we have two valid lists equating with each other (see below):

```
aListCopy = aList;
```



Notes on Class Design

- ▶ `constructor.cpp`
 - ▶ Illustration on when and how to make constructor calls
- ▶ **If a class allocates memory at run time using `new`, then it should provide**
 - ▶ A destructor
 - ▶ An assignment operator
 - ▶ A copy constructor

Dynamic-Allocation for List Class

- ▶ Changes required in data members
 - ▶ Eliminate const declaration for **CAPACITY**
 - ▶ Add variable data member to store capacity specified by client program
 - ▶ Change array data member to a pointer
 - ▶ Constructor requires considerable change
- ▶ Little or no changes required for
 - ▶ **empty()**
 - ▶ **display()**
 - ▶ **erase()**
 - ▶ **insert()**
- ▶ **DList.h, DList.cpp** and **dlisttester.cpp**
- ▶ Another more advanced implementation with exception handling and template (incomplete as it is without copy constructor and assignment operator)
 - ▶ `l1ist.h, l1ist.cpp, xcept.h`

Futher Improvements to Our List Class

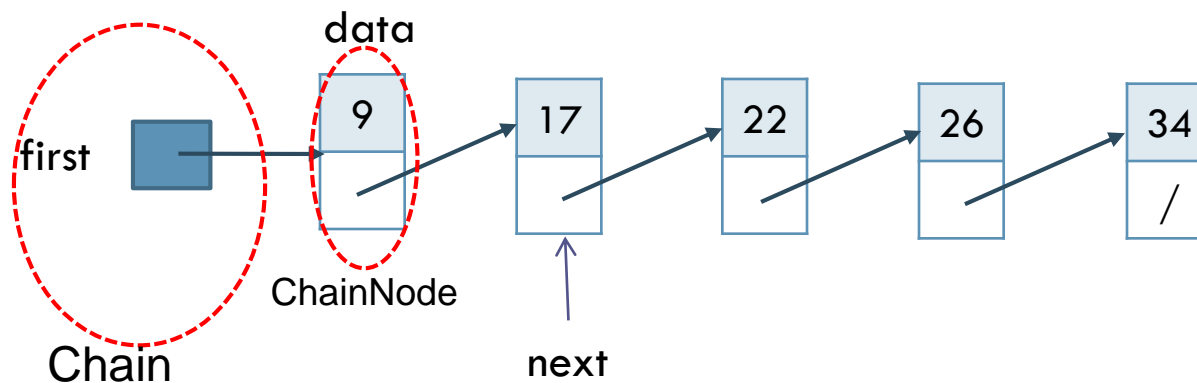
- ▶ **Problem 1: Array used has fixed capacity**
 - ▶ If a larger array is needed during `insert()`, we should allocate a larger array → Allocate (e.g., to double the size), copy smaller array to the new one
 - ▶ To conserve memory, if there are very few elements (say less than 25% full after `erase()`), we can new a smaller array (e.g., half the size), copy the content over and deallocate
 - ▶ Or use linked list
- ▶ **Problem 2: Class bound to one type at a time**
 - ▶ Creating multiple List classes with differing types is difficult
 - ▶ Use class template (later)

Inefficiency of Array-Implemented List

- ▶ **insert()** and **erase()** functions inefficient for dynamic lists
 - ▶ Those sizes that change frequently
 - ▶ Those with many insertions and deletions
- ▶ We look for an alternative implementation
 - ▶ chained nodes using linked list
- ▶ **Linked list**
 - ▶ Remove requirement that list elements be stored in consecutive location
 - ▶ But then need a "link" that connects each element to its successor

Linked List Nodes

- ▶ **ChainNode class**
 - ▶ data
 - ▶ stores an element of type T
 - ▶ next
 - ▶ stores link/pointer to next element
 - ▶ when there is no next element, NULL value
- ▶ **Chain class**
 - ▶ Make use of ChainNode
 - ▶ The nodes are indexed from 1 onwards



Using C++ Pointers and Classes

- ▶ To Implement linked list:

```
template <class T> class Chain;
```

```
template <class T>
class ChainNode {
    friend Chain<T>;
private:
    T data;
    ChainNode<T> *link;
};
```

- ▶ Allows Chain to directly access private members
 - ▶ ChainNode is open only to Chain access
 - ▶ Using accessor and mutator, or making the data members `public`, give access to all other classes
- ▶ The definition of a `ChainNode` is recursive (or self-referential)
- ▶ The next member is defined as a pointer to a `ChainNode`
- ▶ `cnode.h`, `chain.h`, `chain.cpp`
 - ▶ Exception handling `xcept.h`

Linked Lists Operations

- ▶ Construction

 - `first = null;`

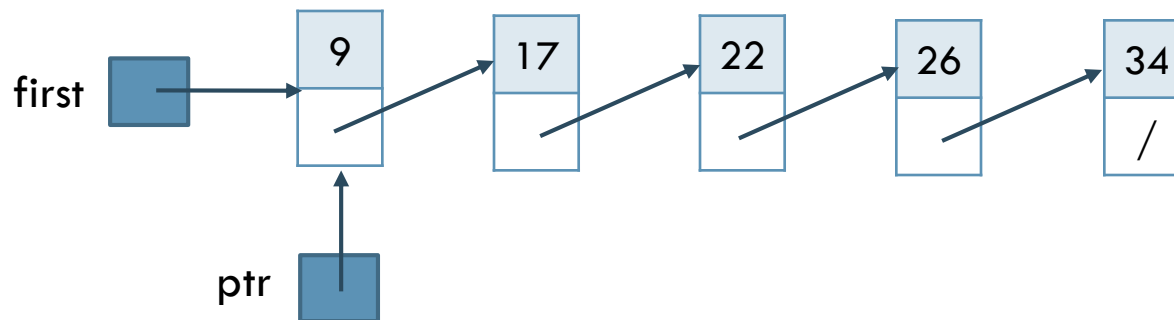
- ▶ Traverse

 - ▶ Initialize a variable `ptr` to point to first node

 - ▶ Continue until `ptr == null`

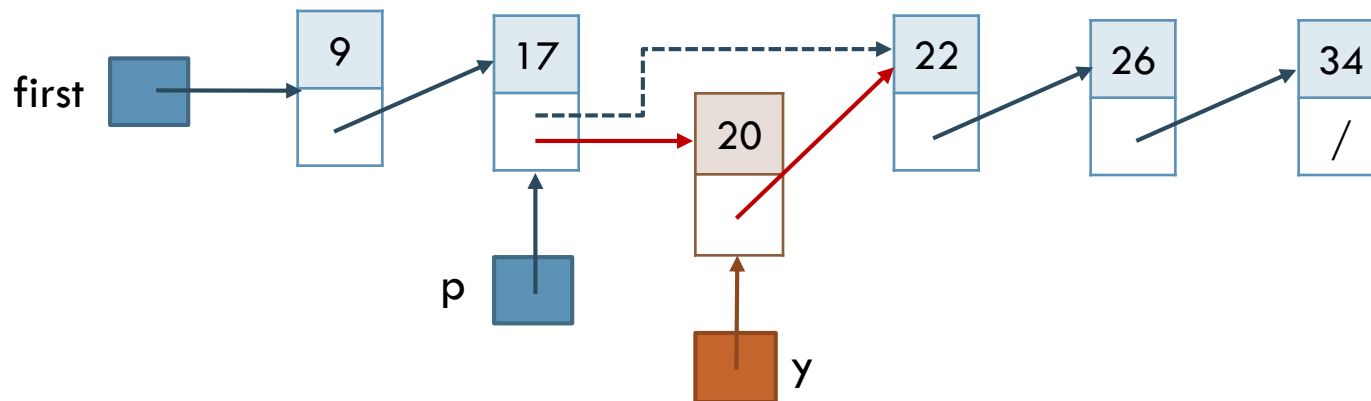
 - ▶ Process data where `ptr` points

 - ▶ Set `ptr = ptr->next`



Operations: Insertion

- ▶ `Insert(int k, const T& x)`
 - ▶ Insert a node with data x *after* the k th node, i.e., becoming the $(k+1)$ th element after the insertion
 - ▶ $0 \leq k \leq \text{size}$,
 - ▶ $k == 0$ means that insertion at the head
 - ▶ $k == \text{size}$ means insertion at the end
- ▶ To insert a node with data 20 after 17
 - ▶ Need address of item before point of insertion
 - ▶ p points to the node containing 17
 - ▶ Get a new node pointed to by y and store 20 in it
 - ▶ Set the next pointer of this new node equal to the next pointer in its predecessor, thus making it point to its successor
 - ▶ Reset the next pointer of its predecessor to point to this new node

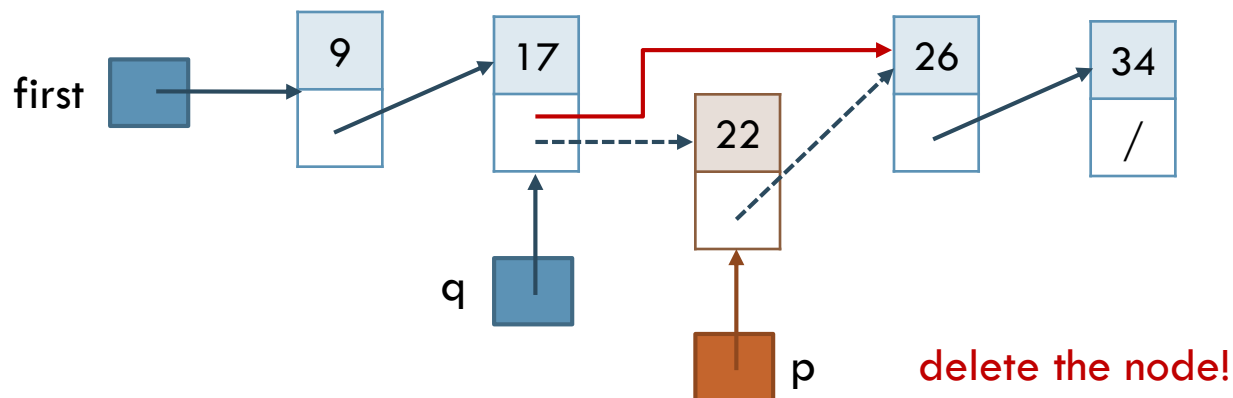


Operations: Insertion

- ▶ Insertion also works at end of list
 - ▶ pointer member of new node set to null
- ▶ Need to worry the special case: Insertion at the beginning of the list
 - ▶ because `p` is a `ChainNode<T>` pointer and cannot point to `first`
 - ▶ `y` sets to `first`
 - ▶ `first` sets to value of `y`
- ▶ In all cases, no shifting of list elements is required!

Operations: Deletion

- ▶ `Delete(int k, T& x)`
 - ▶ Delete the k th node and return its data to x , where $1 \leq k \leq \text{size}$
- ▶ Delete node containing 22 from list
 - ▶ Suppose p points to the node to be deleted
 - ▶ q points to its predecessor (the node with data 17)
- ▶ Do a bypass operation:
 - ▶ Set the next pointer in the predecessor to point to the successor of the node to be deleted (the 26)
 - ▶ Deallocate the node being deleted (the 22).
- ▶ Need to worry the special case: Delete the first element



Linked Lists: Advantages

- ▶ Access any item as long as external link to first item is maintained
- ▶ Insert a new item without shifting
- ▶ Delete an existing item without shifting
- ▶ Can expand/contract as necessary

Linked Lists Overhead

- ▶ **Overhead of links:**
 - ▶ used only internally, pure overhead
- ▶ **Must provide**
 - ▶ destructor
 - ▶ copy constructor and assignment operation
- ▶ **Access of n th item now is less efficient**
 - ▶ must go through first element, and then second, and then third, etc.
- ▶ **No longer have direct access to each element of the list**
 - ▶ Many sorting algorithms need efficient direct access

Variant of Linked Lists: A Clever Trick

- ▶ **Note the loop inefficiencies in insert, delete and search operations**
 - ▶ Insert: the checking of the special case of head insertion
 - ▶ Delete: the checking of special case of head deletion
 - ▶ Search: Need to be careful of the unsuccessful case when the pointer will traverse to NULL
- ▶ **To simplify these operations, we may keep a header node. The header node is a dummy node that does not have to store any data.**
 - ▶ An empty list contains only the header node.
- ▶ **In a non-empty list, nodes storing data follow the header node.**
 - ▶ Saving on comparison and handling special cases in insert, delete and search

Circular Linked List

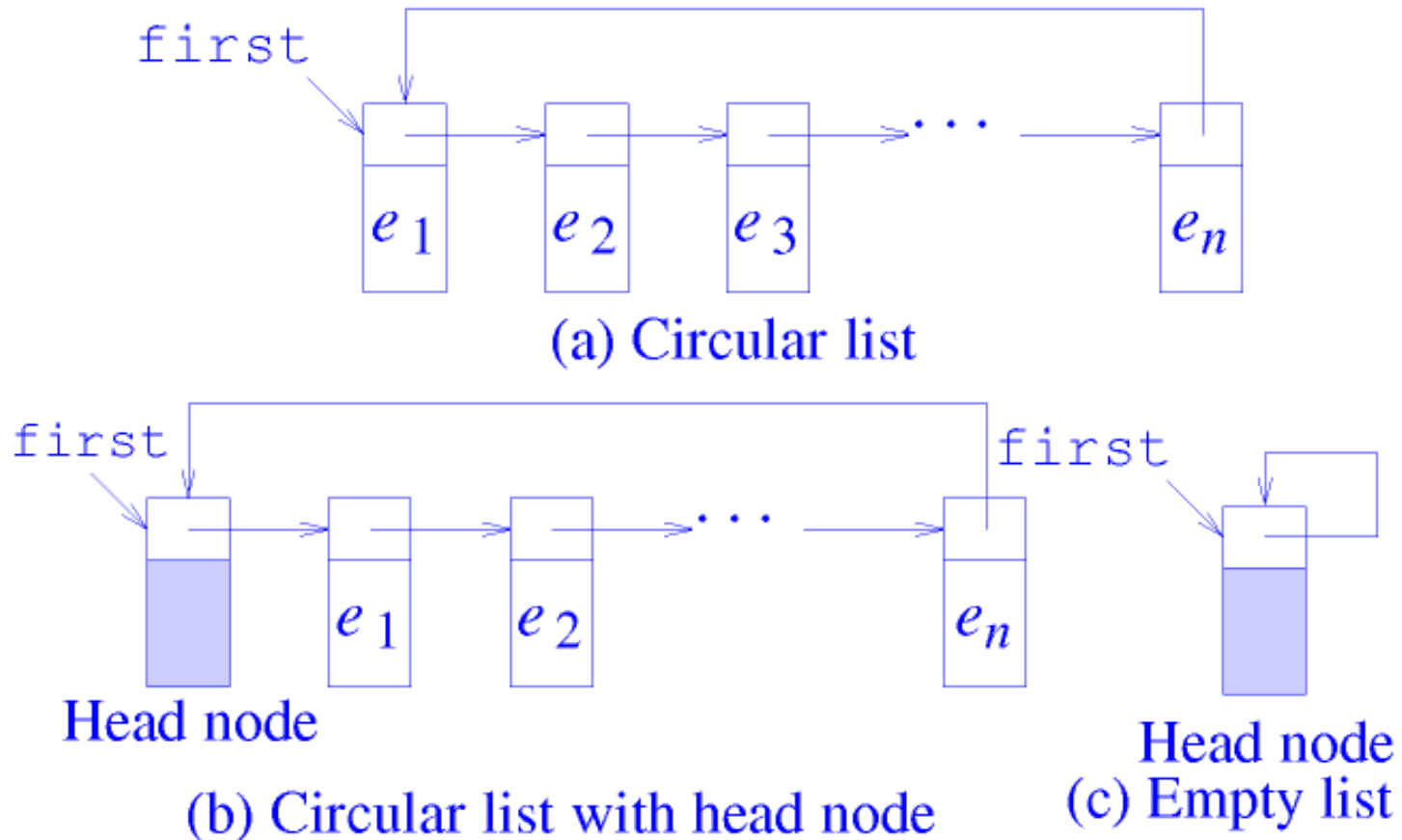


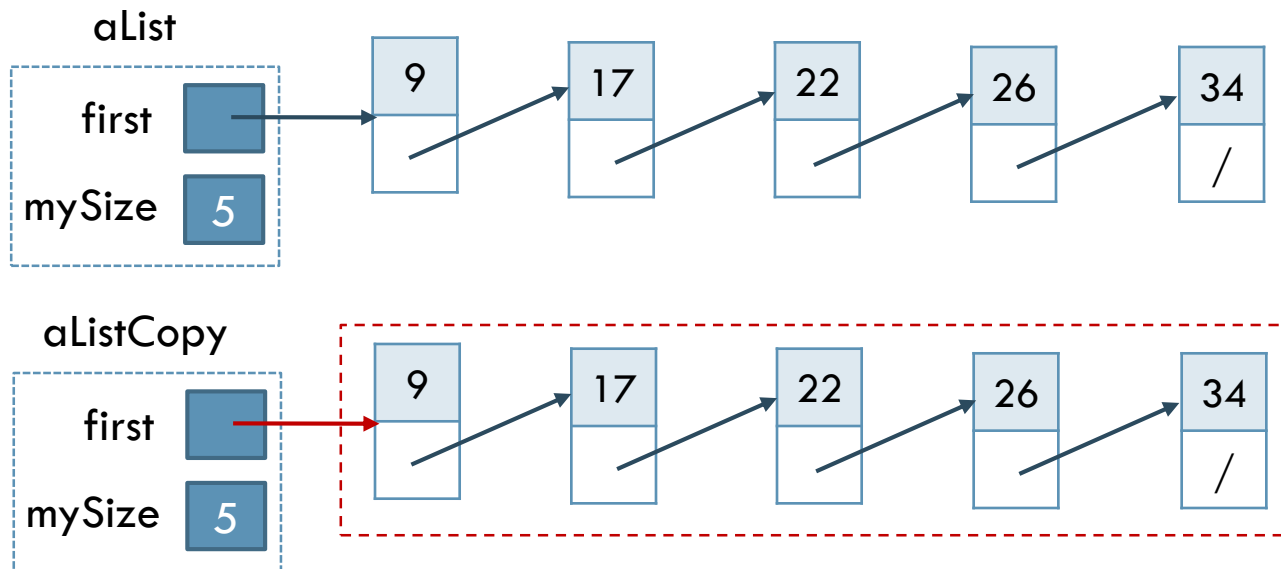
Figure 3.8 Circular linked lists

Efficient Search on a Circular Linked List with a Header Node (Stamp x at the Header)

```
template<class T>
int CircularList<T>::Search(const T& x) const
{ // Locate x in a circular list with head node.
  ChainNode<T> *current = first->link;
  int index = 1; // index of current
  first->data = x; // put x in head node
  // search for x
  while (current->data != x) {
    current = current->link;
    index++;
  }
  // are we at head?
  return ((current == first) ? 0 : index);
}
```

Function Members for Linked-List Implementation

- ▶ Copy constructor for deep copy
 - ▶ By default, when a copy is made of a List object, it only gets the head pointer
 - ▶ Copy constructor will make a new linked list of nodes to which copy will point
- ▶ Assignment operation (=) also needs to be done similarly



Linked Implementation of Sparse Polynomials

- ▶ Consider a polynomial of degree n

- ▶ Can be represented by an array

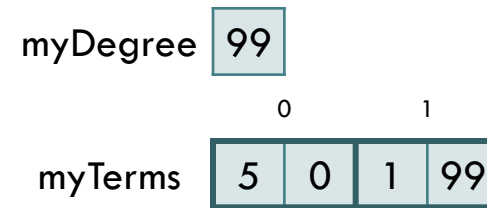
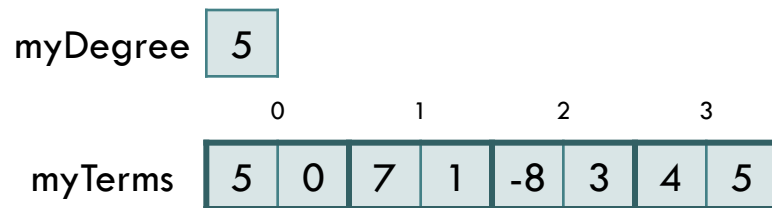
myDegree	5									
	0	1	2	3	4	5	6	7	8	9
myCoeffs	5	7	0	-8	0	4	0	0	0	0

- ▶ For a sparse polynomial this is not efficient

myDegree	99																		
	0	1	2	3	4	5	6	7	8	9	...	95	96	97	98	99			
myCoeffs	5	0	0	0	0	0	0	0	0	0	...	0	0	0	0	1			

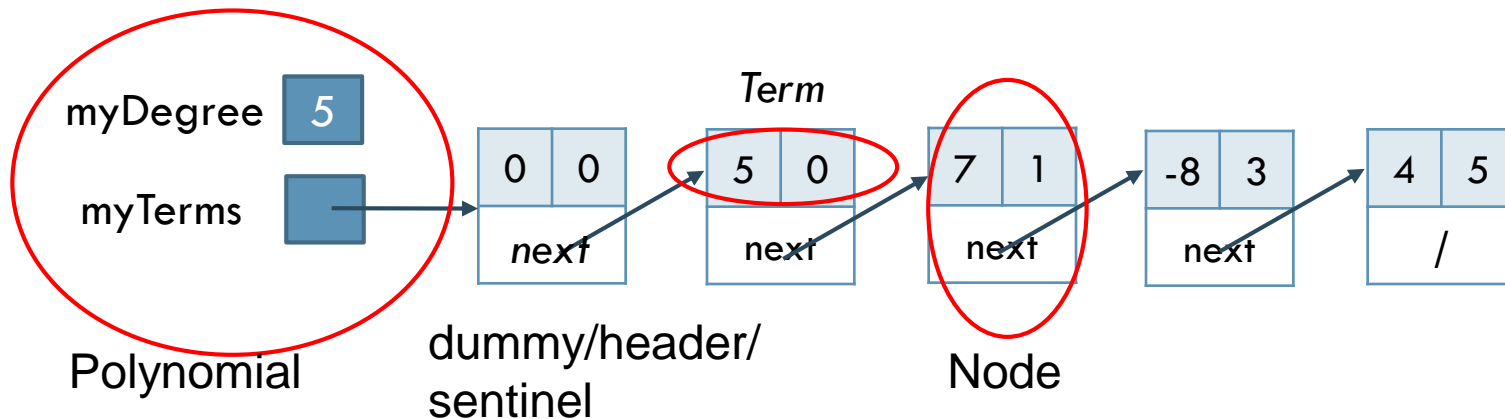
Linked Implementation of Sparse Polynomials

- ▶ We could represent a polynomial by an array of ordered pairs of linked nodes { (coef, exponent) ... }



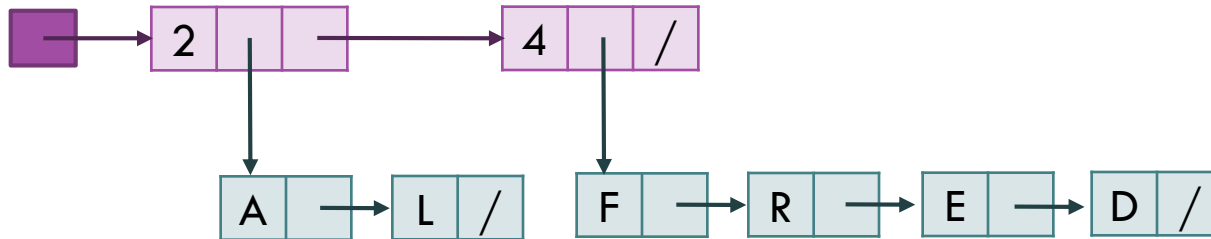
Linked Implementation of Sparse Polynomials

- ▶ Linked list of these ordered pairs provides an appropriate solution
- ▶ Now whether sparse or well populated, the polynomial is represented efficiently
- ▶ Polynomial class (**Polynomial.h**)
 - ▶ Type parameter **CoefType**
 - ▶ **Term** and **Node** are inner private classes for internal use and access only
 - ▶ Used to create internal data structure of a linked node (for internal access only)

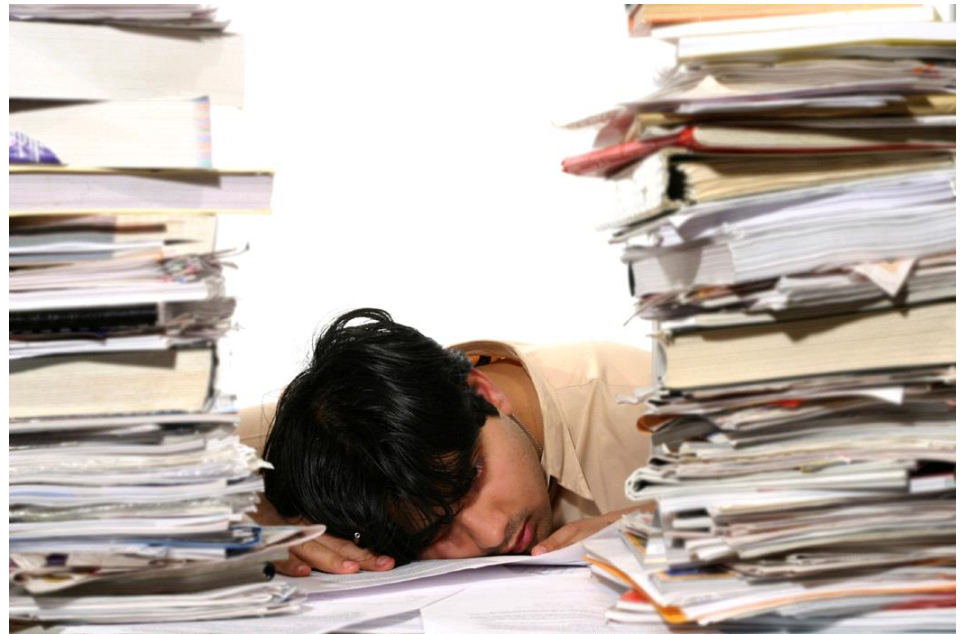


Generalized Lists

- ▶ Examples so far have had atomic elements
 - ▶ The nodes are not themselves lists
- ▶ Consider a linked list of strings
 - ▶ The strings themselves can be linked lists of characters



Stacks



Introduction to Stacks

- ▶ Consider a pile of books
 - ▶ New item is always placed on the top of the pile
 - ▶ Retrieve an item only from the top
- ▶ We seek a way to represent and manipulate this in a computer program – this is a stack
- ▶ A stack is a last-in-first-out (LIFO) data structure
- ▶ Adding an item
 - ▶ Referred to as pushing it onto the stack
- ▶ Removing an item
 - ▶ Referred to as poppping it from the stack



A Stack

▶ Definition:

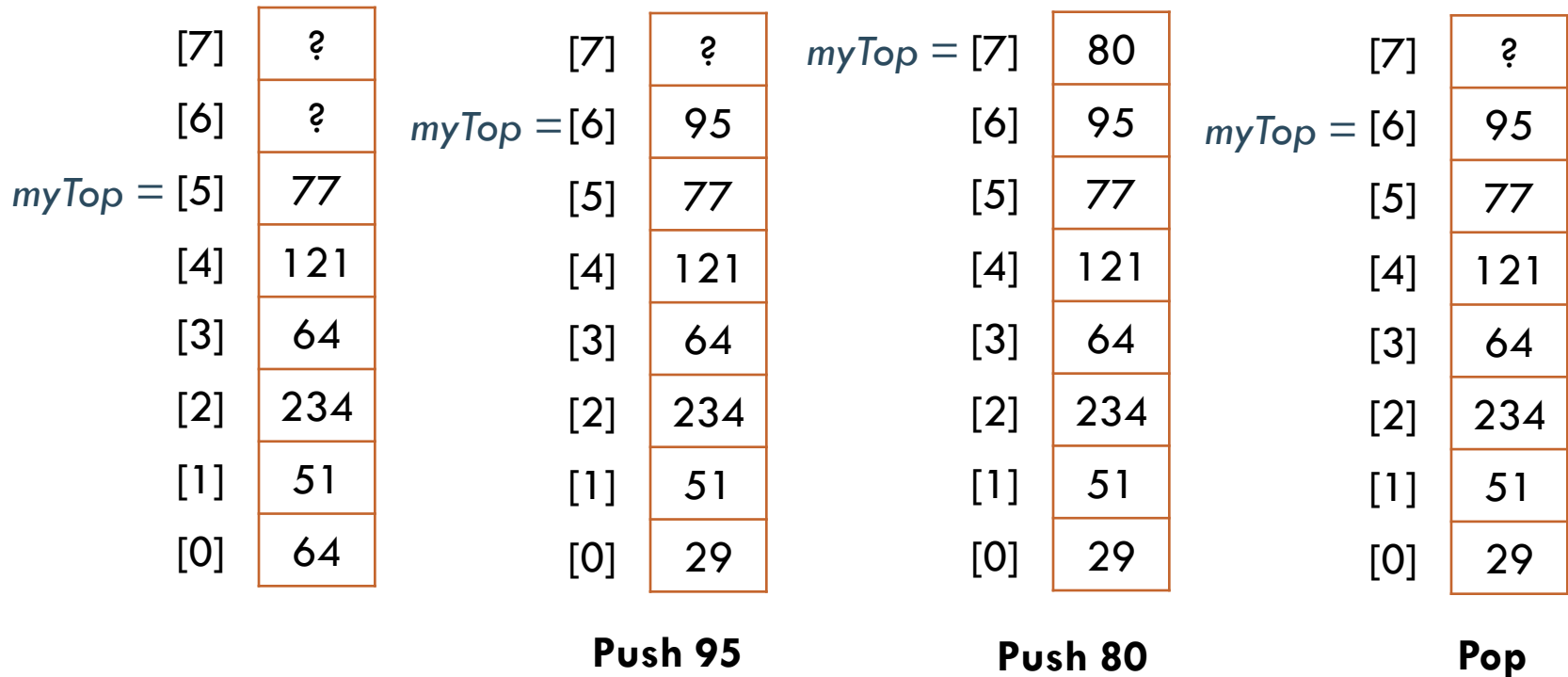
- ▶ An ordered collection of data items
- ▶ Can be accessed at only one end (the top)

▶ Operations:

- ▶ construct a stack (usually empty)
- ▶ check if it is empty
- ▶ Push: add an element to the top
- ▶ Top: examine/peek the top element
- ▶ Pop: remove the top element (usually no input parameter and return type)

Selecting Storage Structure

- ▶ A better approach is to let position 0 be the bottom of the stack
 - ▶ An integer to indicate the top of the stack (**Stack.h**)



Implementing Operations

- ▶ **Constructor**
 - ▶ Compiler will handle allocation of memory
- ▶ **Empty**
 - ▶ Check if value of `myTop == -1`
- ▶ **Push (if `myArray` not full)**
 - ▶ Increment `myTop` by 1
 - ▶ Store value in `myArray[myTop]`
- ▶ **Top**
 - ▶ If stack not empty, return `myArray[myTop]`
- ▶ **Pop**
 - ▶ If array not empty, decrement `myTop`
- ▶ **`Stack.h`, `Stack.cpp` and `driver.cpp`**

Example Program

- ▶ Consider a program to do base conversion of a number (ten to two)
 - ▶ $26 = (11010)_2$
- ▶ It assumes existence of a **Stack** class to accomplish this
 - ▶ Demonstrates **push**, **pop**, and **top**
 - ▶ Push the remainder onto a stack and pop it up one by one
- ▶ Can be written easily using recursion
 - ▶ **n2b.cpp**
- ▶ **Stack.h**, **Stack.cpp** and **BaseConversion.cpp**

$$\begin{array}{r} 2 \overline{) 26} \\ \underline{26} \\ 0 \end{array}$$
$$\begin{array}{r} 2 \overline{) 13} \dots 0 \\ \underline{12} \\ 1 \end{array}$$
$$\begin{array}{r} 2 \overline{) 6} \dots 1 \\ \underline{4} \\ 2 \end{array}$$
$$\begin{array}{r} 2 \overline{) 3} \dots 0 \\ \underline{2} \\ 1 \end{array}$$
$$\begin{array}{r} 2 \overline{) 1} \dots 1 \\ \underline{0} \\ 1 \end{array}$$

0 ... 1

Stack Applications: Parenthesis Matching (Run)

```
cssu5:> a.out
Type an expression of length at most 100
(a+b)-4
The pairs of matching parentheses in
(a+b)-4
are
1 5
cssu5:> a.out
Type an expression of length at most 100
((a+b)*c+d)+e)
The pairs of matching parentheses in
((a+b)*c+d)+e)
are
2 6
1 11
No match for right parenthesis at 14
cssu5:> a.out
Type an expression of length at most 100
(((a+b)
The pairs of matching parentheses in
(((a+b)
are
3 7
No match for left parenthesis at 2
No match for left parenthesis at 1
```

Stack Applications: Parenthesis Matching

- ▶ Match the left and right parentheses in a character string
- ▶ Scan the input string character by character from left to right. Positions are numbered from 1 onwards
- ▶ Push the *position* of the opening (or left) parenthesis into the stack
- ▶ If closing (or right) parenthesis is encountered, it is matched to the left parenthesis at the top of the stack:
 - ▶ pop the stack; and
 - ▶ cout the matched parenthesis positions
- ▶ Error occurs upon
 - ▶ pop error → “No match for right parenthesis”
 - ▶ Non-empty stack at the end of the operations → “No match for left parenthesis”
- ▶ **paren.cpp**

Application of Stacks: Infix to postfix

- ▶ Consider the arithmetic statement in the assignment statement:
 $x = a * b + c$
- ▶ This is "infix" notation: the operators are between the operands

RPN or Postfix Notation

- ▶ Reverse Polish Notation (RPN) = Postfix notation
- ▶ Most compilers convert an expression in *infix* notation to *postfix*
- ▶ The operators are written after the operands
 - ▶ So “a * b + c” becomes “a b * c +”
 - ▶ Easier for compiler to work on loading and operation of variables
- ▶ **Advantage: expressions can be written without parentheses**

Postfix and Prefix Examples

Infix	RPN (Postfix)	Prefix
$A + B$	$A B +$	$+ A B$
$A * B + C$	$A B * C +$	$+ * A B C$
$A * (B + C)$	$A B C + *$	$* A + B C$
$A - (B - (C - D))$	$A B C D ---$	$-A-B-C D$
$A - B - C - D$	$A B-C-D-$	$---A B C D$

Prefix : Operators come before the operands

Rules of Conversion

- ▶ Always write out operand
- ▶ Always push '(' into the stack
- ▶ An operator can only be pushed into a stack if it is of *higher* priority than the one below
 - ▶ $*, /$ is higher than $+, -$
 - ▶ If not, keep popping the stack until it is so, hit a '(', or the stack is empty
- ▶ Upon the operator ')', always pop the stack till '('

Stack Algorithm (postfix.cpp)

1. Initialize an empty stack of operators
2. While no error and not end of expression
 - a) Get next input "token" from infix expression, where a token is a constant/variable/arithmetic operator/parenthesis
 - b) switch(token):
 - i. "(" : push onto stack
 - ii. ")" : pop the stack and display the element until "(" occurs, do not display the "("
 - iii. operator:
if the operator has *higher* priority than the top of stack
 push token onto the stack
else
 pop the stack and display it
 if the top is not "(", repeat comparison of the operator with the top of the stack
 - iv. operand: display it
3. End of infix reached: pop and display stack items until empty

Evaluating RPN Expressions (Similar to Compiler Operations)

"By hand" (Underlining technique):


1. Scan the expression from left to right to find an operator
2. Locate ("underline") the last two preceding operands and combine them using this operator
3. Repeat until the end of the expression is reached

Example:

```
2 3 4 + 5 6 - - *  
→ 2 3 4 + 5 6 - - *  
→ 2 7 5 6 - - *  
→ 2 7 5 6 - - *  
→ 2 7 -1 - *  
→ 2 7 -1 - *  
→ 2 8 *  
→ 2 8 *  
→ 16
```

Evaluating RPN Expressions

By using a stack algorithm

1. Initialize an empty stack
2. Repeat the following until the end of the expression is encountered
 - a) Get the next token (const, var, operator) in the expression
 - b) Operand – push onto stack
Operator – do the following
 - i. Pop 2 values from stack 
 - ii. Apply operator to the two values
 - iii. Push resulting value back onto stack
3. When end of expression encountered, value of expression is the *only* number left in stack; otherwise the expression is in error.

Note: if only 1 value on stack, this is a pop error, i.e., an invalid RPN expression

Evaluation of Postfix

2 4 * 9 5 + -	2	Push 2 onto the stack
4 * 9 5 + -	4 2	Push 4 onto the stack
* 9 5 + -	8	Pop 4 and 2 from the stack , multiply, and push the result 8 back
9 5 + -	9 8	Push 9 onto the stack
5 + -	5 9 8	Push 5 onto the stack
+ -	14 8	Pop 5 and 9 from the stack, add, and push the result 14 back
-	-6	Pop 14 and 8 from the stack, subtract, and push the result -6 back
(end of strings)	-6	Value of expression is on top of the stack

Queues

Introduction to Queues

- ▶ A queue is a waiting line – seen in daily life
 - ▶ A line of people waiting for a bank teller
 - ▶ A line of cars at a toll booth
 - ▶ "This is the captain, we're 5th in line for takeoff"

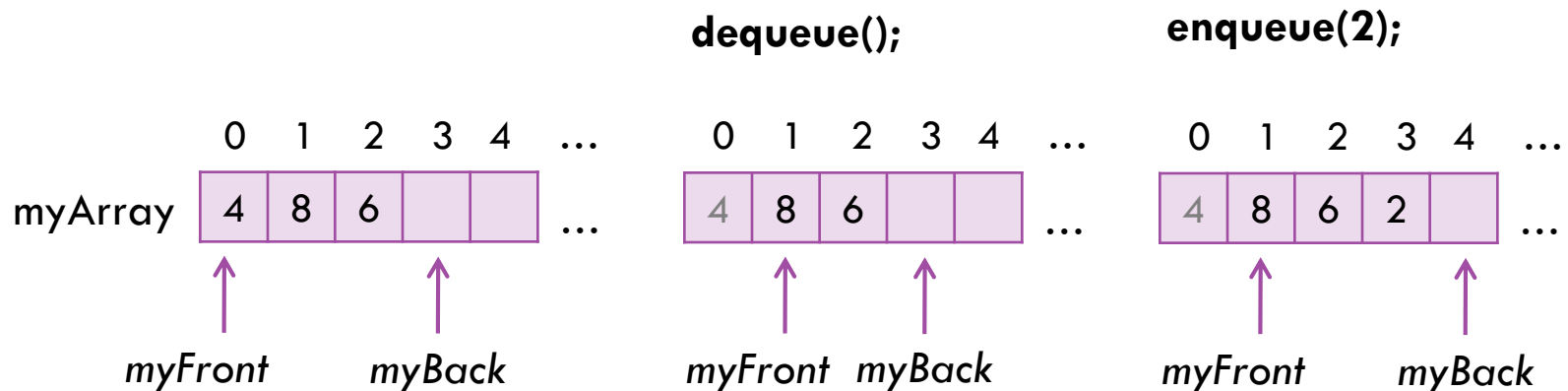


The Queue as an ADT

- ▶ A queue is a sequence of data elements
- ▶ In the sequence
 - ▶ Items can be removed only at the front
 - ▶ Items can be added only at the other end, the back
- ▶ Basic operations
 - ▶ Construct a queue
 - ▶ Check if empty
 - ▶ Enqueue (add element to back)
 - ▶ Front (retrieve value of element from front)
 - ▶ Dequeue (remove element from front)

Array-Based Queue Class

- ▶ Consider an array in which to store a queue
- ▶ Additional variables needed
 - ▶ **myFront** and **myBack**



Array-Based Queue Class

▶ Problems

- ▶ We quickly "walk off the end" of the array
- ▶ Shift array elements? Inefficient!

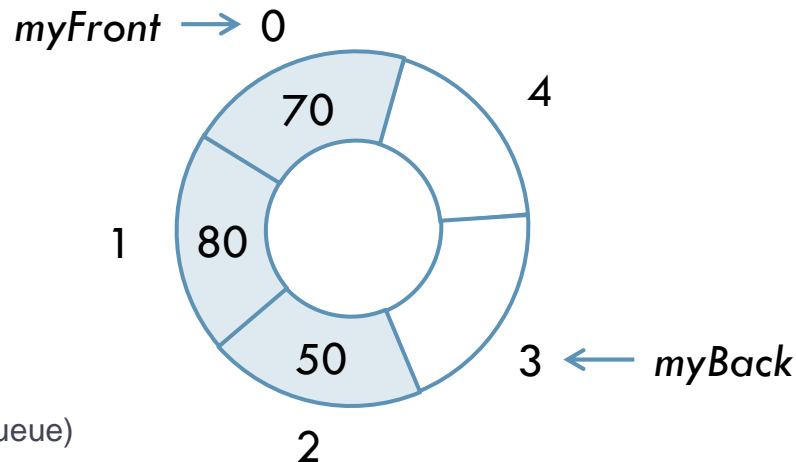
▶ A more efficient solution: Circular array/queue

- ▶ `myBack` *always* points to an *empty* slot to put the next item in
 - ▶ `myBack` cannot point to a *valid* item because we need a way to distinguish an empty queue (if we have an empty queue, where should `myBack` point to?)
- ▶ An empty queue is indicated by `myBack == myFront`
- ▶ A full queue is indicated by `(myBack+1) % queue_size == myFront`
- ▶ Because of the above, the capacity of the queue is one LESS the actual array size

enqueue(70);

enqueue(80);

enqueue(50);

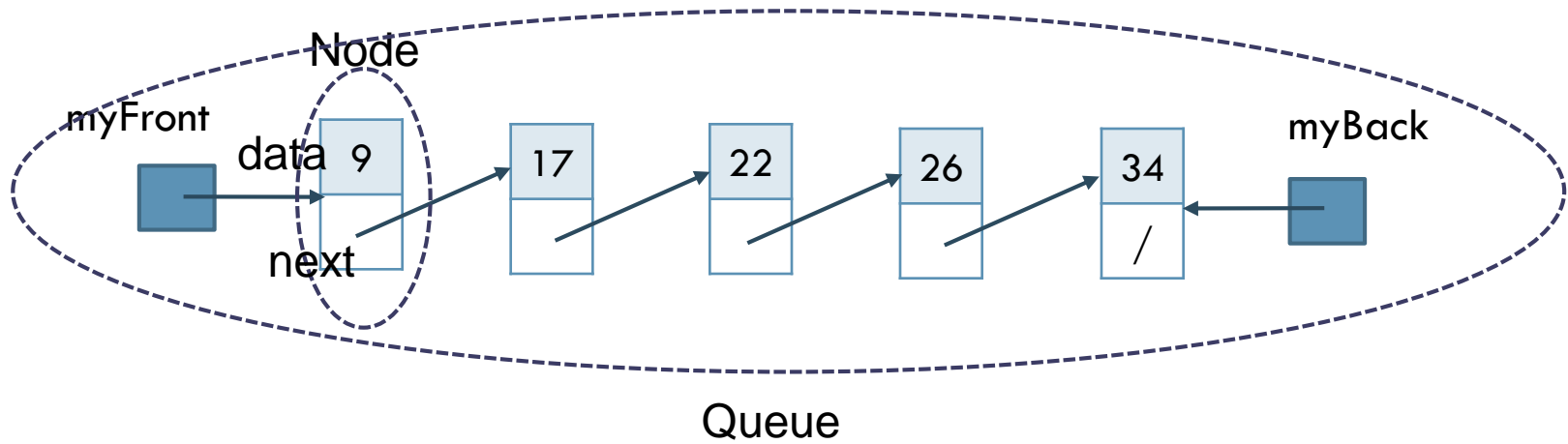


Array-Based Queue Class

- ▶ Using a static array (`Queue.h` and `Queue.cpp`)
 - ▶ `QUEUE_CAPACITY` specified
 - ▶ Enqueue increments `myBack` using mod operator, checks for full queue
 - ▶ Dequeue increments `myFront` using mod operator, checks for empty queue
- ▶ Instead of `myBack`, a more clever implementation is to keep n , the number of valid elements in the queue
 - ▶ $n == 0$ means empty queue
 - ▶ $n == \text{myCapacity}$ means full queue
 - ▶ In this case, you can use the full array.
- ▶ Similar problems as list and stack
 - ▶ Fixed size array can be specified either too large or too small
- ▶ Dynamic array design allows sizing of array for multiple situations
 - ▶ Need a destructor, copy constructor and assignment (`=`) operator
 - ▶ `myCapacity` determined at run time

Linked Queues

- ▶ Even with dynamic allocation of queue size
 - ▶ Array size is still fixed
 - ▶ Cannot be adjusted during run of program
- ▶ Could use linked list to store queue elements
 - ▶ Can grow and shrink to fit the situation
 - ▶ No need for upper bound (**myCapacity**)



Linked Queues

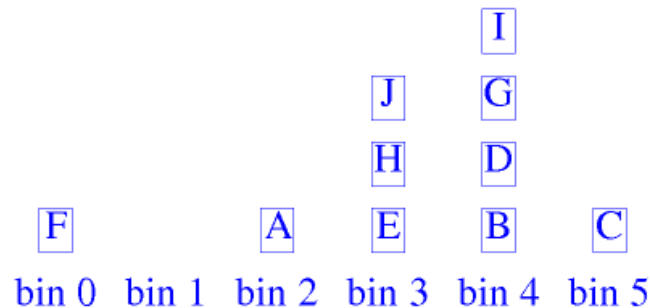
- ▶ Constructor initializes
 - ▶ `myFront = null;`
 - ▶ `myBack = null;`
- ▶ Front
 - ▶ return `myFront->data`
- ▶ Dequeue
 - ▶ Delete first node (watch for empty queue)
- ▶ Enqueue
 - ▶ Insert node at end of list (watch for empty queue)
- ▶ **LQueue.h, LQueue.cpp, driver.cpp**

Queue Application: Bin (or Bucket) Sort

- ▶ The nodes are placed into bins, each bin contains nodes with the same score
- ▶ Then we combine the bins to create a sorted chain
- ▶ Note that it does not change the relative order of nodes that have the same score, the so-called *stable sort*.



(a) Input chain



(b) Nodes in bins



(c) Sorted chain

Radix Sort

- ▶ Sort, linear time, m integers in the range 0 through $r^d - 1$, where r is a constant (the radix).
- ▶ We decompose the numbers using the radix r , and then sort by digits
- ▶ For example, with $r = 10$, we can use *binsort* with the number of bin equal to r to sort the digit one by one in *increasing* significance

Radix Sort Illustrated (with $r = 10$ and $d = 3$)



(a) Input chain



(b) Chain after sorting on least significant digit



(c) Chain after sorting on 2nd least significant digit



(d) Chain after sorting on most significant digit

Figure 3.17 Radix sort with $r = 10$ and $d = 3$

Radix Sort: Another Example

Think of each element in your input, $A_1, A_2, \dots, A_i, \dots, A_{n-1}, A_n$ as being composed of several digits. Such as:

$$\text{Element } A = D_d D_{d-1} \dots D_i \dots D_2 D_1$$

$$\begin{array}{l} \text{So, } 30485 \end{array} \left\{ \begin{array}{l} d = 5 \\ D_1 = 5 \\ D_2 = 8 \\ D_3 = 4 \\ D_4 = 0 \\ D_5 = 3 \end{array} \right.$$
$$345 \left\{ \begin{array}{l} d = 5 \\ D_1 = 5 \\ D_2 = 4 \\ D_3 = 3 \\ D_4 = 0 \\ D_5 = 0 \end{array} \right.$$

What does this mean? Our elements have a maximum size, defined by d .

Basic Idea

Sort by least-significant digit first, then next digit, and so on.

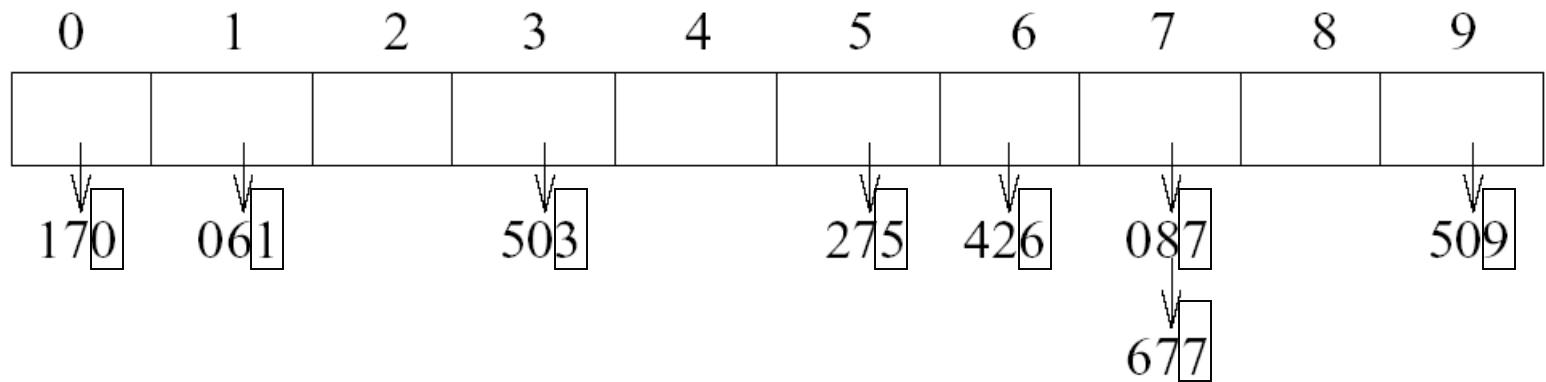
That is:	25 6 ,	23 4 ,	09 0 ,	30 4	First (least significant) digit
	2 5 6,	2 3 4,	0 9 0,	3 0 4	Second digit
	2 56,	2 34,	0 90,	3 04	Third (most significant) digit

Example. Number composed of 3 digits

(digit is [0 to 9])

Eg. 275, 087, 426, 061, 509, 170, 677, 503

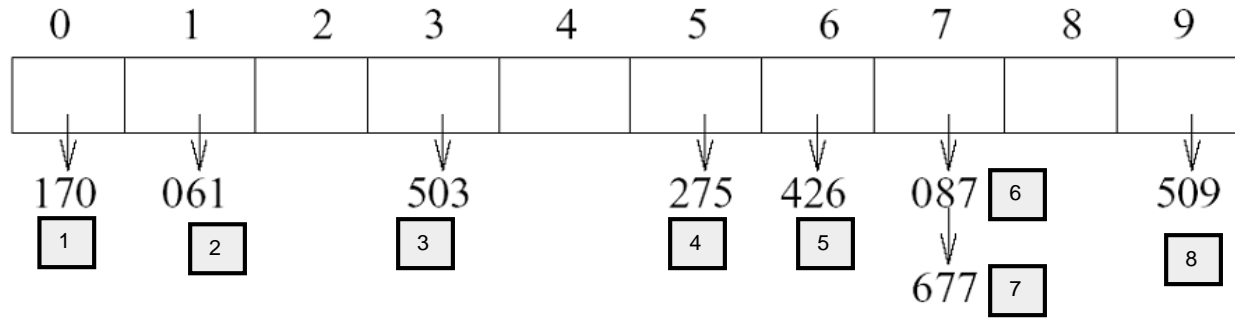
1st pass



First pass sorting by first significant digit.

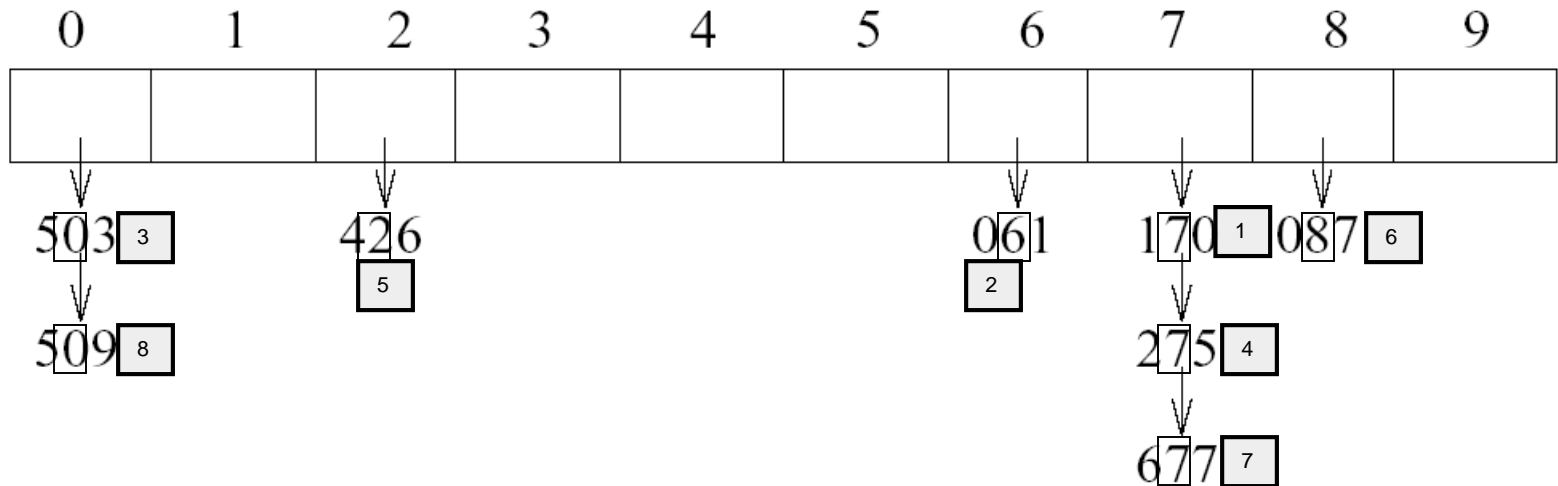
Example: r=10 (base 10)

1st pass



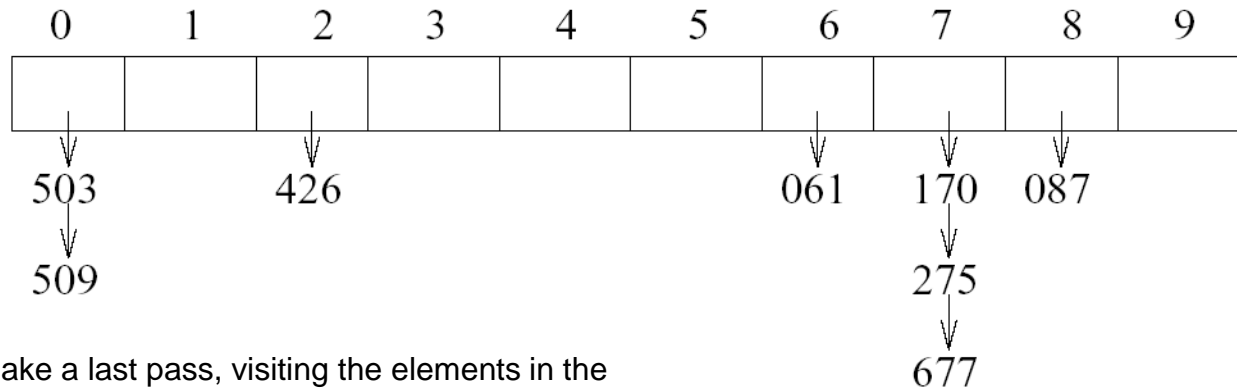
Now, make a second pass, visiting the elements in the order they appear after the first pass.

2nd pass



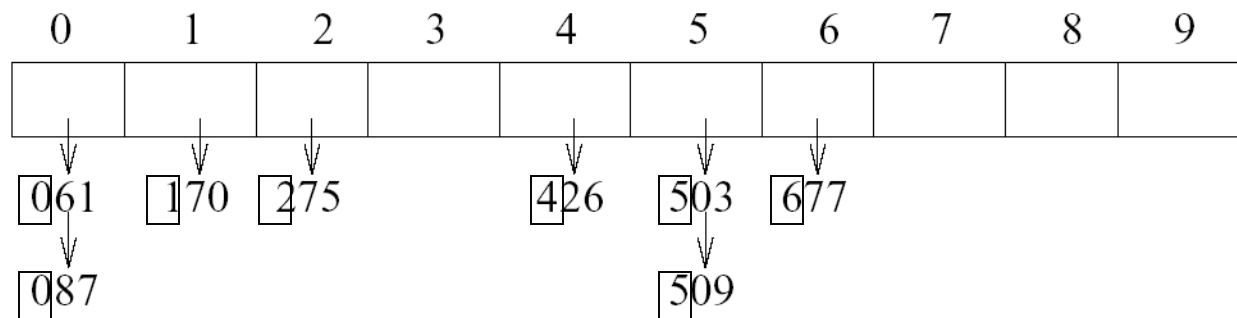
Last Pass

2nd pass



Now, make a last pass, visiting the elements in the order they appear after the second pass.

3rd pass



Look! Sorted!

We'll use a queue data-structure to help us out.

Algorithm *RadixSort*(A, n, d)

```
----- 1. for  $0 \leq p \leq 9$ 
2.     do  $Q[p] :=$  empty queue;
3.  $D := 1$ ;
4. for  $1 \leq k \leq d$ 
5.     do
6.          $D := 10 * D$ ;
7.         for  $0 \leq i < n$ 
8.             do  $t := (A[i] \bmod D) \operatorname{div} (D/10)$ ;
9.                 enqueue( $A[i], Q[t]$ );
10.         $j := 0$ ;
11.        for  $0 \leq p \leq 9$ 
12.            do while  $Q[p]$  is not empty
13.                do  $A[j] :=$  dequeue( $Q[p]$ );
14.                     $j := j + 1$ ;
```

Queues

r=10 (radix)

0

Look at first digit D_1

1

2

For each $A[i] = D_d D_{d-1} \dots D_i \dots D_2 D_1$

3

find $A[i]$'s appropriate queue

4

5

6

7

8

9

Algorithm RadixSort(A, n, d)

```

1. for  $0 \leq p \leq 9$ 
2.   do  $Q[p] :=$  empty queue;
3.  $D := 1$ ;
4. for  $1 \leq k \leq d$ 
5.   do
6.      $D := 10 * D$ ;
7.     for  $0 \leq i < n$ 
8.       do  $t := (A[i] \bmod D) \text{ div } (D/10)$ ;
9.         enqueue( $A[i], Q[t]$ );
10.     $j := 0$ ;
11.    for  $0 \leq p \leq 9$ 
12.      do while  $Q[p]$  is not empty
13.        do  $A[j] :=$  dequeue( $Q[p]$ );
14.           $j := j + 1$ ;

```

$A_1, A_2, A_3 \dots A_i \dots A_{n-1}, A_n$

Array input

Set up 10 empty queue for each possible value of a digit [0-9]

Finding digit D_i From Key

- ▶ We want to find the i^{th} digit in our key A

$$A = D_d D_{d-1} \dots D_i \dots D_2 D_1$$

Get Digit D_i

$$\text{Let } D = 10^i$$

$$\text{Then } D_i = (A \bmod D) \text{ div } (D/10)$$

Example. $A=30487$ we want D_3

$$i=3, D=1000$$

$$A \bmod 1000 = 487$$

$$487 \text{ div } (D/10)$$

$$487 \text{ div } (100) = 4$$

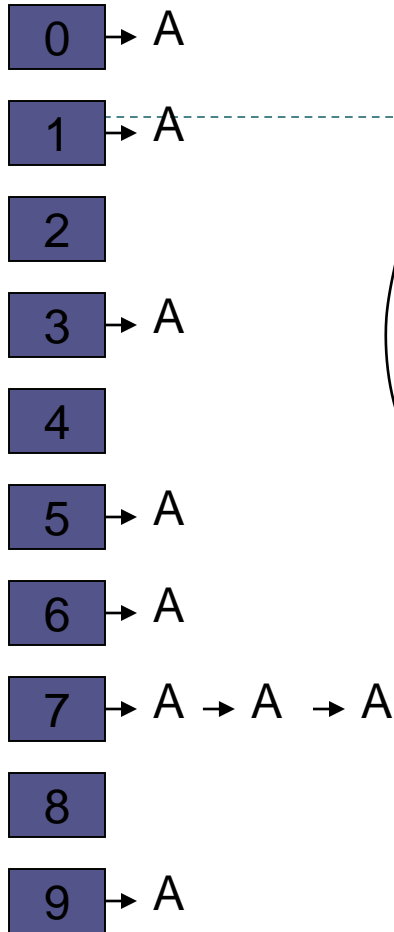
(first)

(second)

result!

mod and div are integer operators

Queues



Look at first digit D_1

For each $A[i] = D_d D_{d-1} \dots D_i \dots D_2 D_1$

$t = (A[i] \bmod D) \text{ div } (D/10)$

$Q[t] \rightarrow \text{enqueue}(A[i])$

Algorithm *RadixSort*(A, n, d)

```

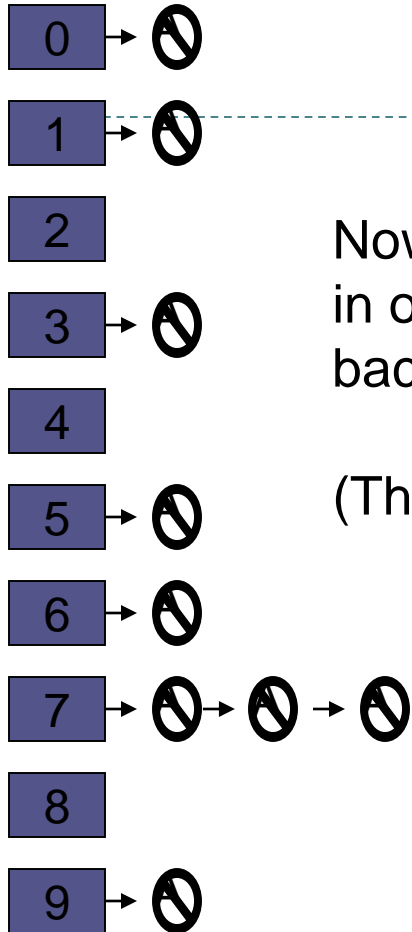
1. for  $0 \leq p \leq 9$ 
2.   do  $Q[p] :=$  empty queue;
3.  $D := 1$ ;
4. for  $1 \leq k \leq d$ 
5.   do
6.      $D := 10 * D$ ;
7.     for  $0 \leq i < n$ 
8.       do  $t := (A[i] \bmod D) \text{ div } (D/10)$ ;
9.         enqueue( $A[i], Q[t]$ );
10.     $j := 0$ ;
11.    for  $0 \leq p \leq 9$ 
12.      do while  $Q[p]$  is not empty
13.        do  $A[j] :=$  dequeue( $Q[p]$ );
14.           $j := j + 1$ ;
```

$A_1, A_2, A_3 \dots A_i \dots A_{n-1}, A_n$

Array input

Set up 10 empty queue for each possible value of a digit [0-9]

Queues



Now de-queue items
in order and place
back in the array.

(This is sorting by Digit D_1)

Algorithm *RadixSort*(A, n, d)

```

1.  for  $0 \leq p \leq 9$ 
2.      do  $Q[p] :=$  empty queue;
3.   $D := 1$ ;
4.  for  $1 \leq k \leq d$ 
5.      do
6.           $D := 10 * D$ ;
7.          for  $0 \leq i < n$ 
8.              do  $t := (A[i] \bmod D) \text{ div } (D/10)$ ;
9.                  enqueue( $A[i], Q[t]$ );
10.          $j := 0$ ;
11.         for  $0 \leq p \leq 9$ 
12.             do while  $Q[p]$  is not empty
13.                 do  $A[j] :=$  dequeue( $Q[p]$ );
14.                      $j := j + 1$ ;
```



Set up 10 empty queue for each
possible value of a digit [0-9]

Repeat procedure
for all digits D_i in our
keys.

In the end, the array will
be sorted!

Algorithm *RadixSort*(A, n, d)

```
1.  for  $0 \leq p \leq 9$ 
2.      do  $Q[p] :=$  empty queue;
3.   $D := 1$ ;
4.  for  $1 \leq k \leq d$ 
5.      do
6.           $D := 10 * D$ ;
7.          for  $0 \leq i < n$ 
8.              do  $t := (A[i] \bmod D) \operatorname{div} (D/10)$ ;
9.                  enqueue( $A[i], Q[t]$ );
10.          $j := 0$ ;
11.         for  $0 \leq p \leq 9$ 
12.             do while  $Q[p]$  is not empty
13.                 do  $A[j] :=$  dequeue( $Q[p]$ );
14.                      $j := j + 1$ ;
```

$A_1, A_2, A_3 \dots A_i \dots A_{n-1}, A_n$

Array input

Number of Steps in binsort

- ▶ Sort m integers
 - ▶ For example, sort $m=1,000$ integers in the range of 0 to 10^6-1
- ▶ We use binsort with range r :
 - ▶ Initialization of array takes r steps;
 - ▶ putting items into the array takes m steps;
 - ▶ and reading out from the array takes another $m+r$ steps
 - ▶ The total sort complexity is then $(2m + 2r)$, which can be much larger than m if r is large

Optimal r

- ▶ 1000 integers in the range of 0 and $10^6 - 1$ are to be sorted using radix sort.
- ▶ We assume that each step takes the same amount of execution or computation time.

Using $r=10^6$ and $r = 1,000$

- ▶ For $r = 10^6$, we can put them into bins and write them out in one pass
 - ▶ 10^6 initiation on bins
 - ▶ 1,000 steps to distribute the numbers into bins
 - ▶ $10^6 + 1000$ to collect the numbers
 - ▶ A total steps of **2,002,000**
- ▶ Using $r = 1,000$
 - ▶ Sort using the least 3 significant decimal digits of each number and use a bin-range equal to 1,000
 - ▶ Initialization of bins: 1,000 steps
 - ▶ 1,000 steps to distribute the numbers into bins
 - ▶ 1,000 steps to sweep the 1000 bins
 - ▶ 1,000 de-queuing steps to collect the numbers
 - ▶ A total of 3,000 steps (excluding the initialization steps)
 - ▶ Repeat the above one more time using the next three decimal digits of each number, which takes another 3,000 steps as given above
 - ▶ Total steps = $1000 + 3000 + 3000 = 7,000$

How about $r=100$ and 10 ?

▶ With $r = 100$

- ▶ Three bin sorts on pairs of decimal digits are performed
- ▶ 100 queue-initialization steps
- ▶ Each of the sorting stage takes 1,000 distribution steps, and $100+1000$ readout. This totals 2,100 steps
- ▶ With 3 passes, total steps = $100 + 3 \times (2100) = 6,400$

▶ With $r = 10$

- ▶ Six bin sorts on 1 decimal digit at each pass are performed
- ▶ 10 queue-initialization steps
- ▶ Each of the sorting stage takes $1000+(10+1000) = 2,010$ steps
- ▶ Total steps = $10 + 6 \times (2010) = 12,070$.

▶ Therefore, radix sort with $r = 100$ is the most efficient

- ▶ Strike a good balance between the number of passes in binsort and the cost of initialization/readout