# 4.4 Circular lists

## Why circular list?

## check whether a pointer *current* points to the last node

### check for (current->link == first)

#### circular list를 사용하는 이유: Figure 4.13(section 4.7.3: circular list representation of polynomials)

### ensure that the link of the last node points to the first node of the list

### insert a new node at the front of the list

### fig 4

#### have to change the link data member of the node x3

#### have to move down the entire length of a list until the last node

## point to the last node rather than to the first

### fig 4

## to insert x at the front



#### assume the existence of class *CircList*

##### contain the private data member *last* that points to the last node

* list, circular list, doubly linked list에 대한 ADT, class, representation를 정의할 수 있을 것

## have to handle the empty list as a special case

### introduce a dummy head node into each circular list



# 4.5 Available Space Lists

## the destructors for chains and circular lists take time linear

### Let av be a static class member of *CircularList<T>*

### available space list or av list = the chain of deleted nodes

### Program 4.15: GetNode( )

### Program 4.16: RetNode( )

### Program 4.17: ~CircularList( ) – assume Figure 4.15 for last

### Figure 4.17: the link changes involved in deleting a circular list

# 4.6 Linked stacks and queues

## when several stacks and queues coexist

### no efficient way to represent them sequentially



## a collection of m stacks and n queues

### no need to shift stacks or queues around to make space

### a collection of m stacks and n queues is obtained:

#### Stack \*stack = new Stack[m];

#### Queue \*queue = new Queue[n];







* the n-stack, m-queue problem
	+ the solution of linked stacks and queues
		- both computationally and conceptually simple
* stack과 queue는 다른 응용에서 재사용되는 중요한 데이타 구조임
	+ implement stacks and queues directly to make them as efficient as possible

# 4.7 Polynomials(구판 내용이 더 논리적)

## illustrate how class Polynomial can be implemented by using the linked list class

# 4.7.1 Polynomial representation

## be able to represent any number of different polynomials within main memory

## define a Polynomial class to implement polynomials

### polynomials is-implemented-by List

#### declare *List* as a data member of the Polynomial

### make the linked list object *poly* a data member of Polynomial

### use *struct* rather than class to define *Term*

#### make the data members of *Term* public

#### any function that has access to a *Term* object also has access to its data members

#### do not violate data encapsulation for Polynomial



# 4.7.2 Adding polynomials

## to add two polynimials a and b, use the list iterators Aiter and Biter



 Figure 4.20

## assume that a and b have m and n terms

### the total number of terms : m + n

### the computing time: O(m+n)

### iterator object를 사용하는 방법을 공부할 것

### Program 4.24:Adding two polynomials

# 4.7.3 Erasing polynomials

#### d(x) = a(x) \* b(x) + c(x)

#### viod func( )

#### {

####  polynomial a, b, c, d, t;

####  cin >> a; // read and create polynomials

####  cin >> b;

####  cin >> c;

####  t = a \* b

####  d = t + c;

####  cout << d;

#### }

## after function *func* has been exited

### need to return the memory occupied by a, b, c, t, d

#### delete only the polynomial class objects and the List<Term> objects by *destructor*

#### ListNode<Term> objects are not deleted

#### not possible to access ListNode objects

#### the memory ListNode objects occupy is lost



## have to define a destructor to delete all objects that are conceptually part of a class



# 4.7.4 Circular list representation of polynomials

## possible to free all the nodes in a list more efficiently by employing a circular list

## an efficient erase algorithm for circular lists

### maintain a chain of nodes(*av* list, available-space list) that have been "deleted"

### if av list is empty, then need to use command *new* to create a new node

### let *av* be a static class member of CircList<Type> of type ListNode<Type>

### use the functions *CircList::GetNode* and *CircList::RetNode* instead of *new* and *delete*





## erase a circular list by function *CircList<Type>::~CircList*



* the variable *first* points to the last node like Figure 4.15

## have to handle the zero

### introduce a head node into each polynomial as a special case during addition and deletion



### remove the test for first == 0 from ~CircList( )

### the exp field of the head node is set to -1

#### no need for additional code to copy the remaining terms as in operator+( )



# 4.8 Equivalence classes

## 4.3.4 Reusing a Class절에서 언급한 내용

### “there are some scenarios where one should not attempt to reuse a class”

### the operations required by the application are complex and specialized

### 1) x = x : reflexive

### 2) x = y, y = x : symmetric

### 3) x = y, y = z, x = z : transitive

## Def) a relation ≡ over a set S is an equivalence relation over S iff it is symmetric, reflexive, transitive over S

### partition the set S into equivalence classes

## the algorithm to determine equivalence classes

### 1) 1st phase

#### read and store the equivalence pairs (i, j)

### 2) 2nd phase

#### begin at 0 and find all pairs of the form (0, j)

#### by transitively, all pairs of the form (j, k) is in the same class as 0



## how to implement the data structures for holding pairs

### 1) using array pairs[n][n]

#### pairs[i][j] = TRUE iff i and j are paired

##### wasteful of space

##### require O(n\*\*2) time, to initialize the array

### 2) consider a linked list to represent each row

#### still need random access to the *i*th row

#### *seq[n]* can be used as the head node of the n lists

## need a mechanism that tells us whether or not object i is yet to be printed

### *out[n]*



## *seq[i]* points to a list of nodes that contains every number directly equivalent to i

## create a stack of nodes, to process the remaining lists which, by transitivity, belong in the same class as i

### accomplished by changing the link data members



while(x) {//process the list

 j = x->data;

 if (out[j] == FALSE) {

 cout<<”,”<<j;

 out[j] = TRUE;

 ListNode \*y = x->link;

 x->link = top;

 top = x;

 x = y;

 }

 else x = x->link;

} //end of while(x)

if (!top) break;

else {

 x = seq[top->data];

 top = top->link;//unstack

}

# 4.10 Doubly linked lists

## difficulty with singly linked lists

### difficult to find the node that precedes p

### deletion of an arbitrary node requires knowing the preceding node

## a doubly linked circular list



## empty doubly linked list with head node



## p == p→llink→rlink == p→rlink→llink

### one can go back and forth with equal ease

## the class definition of a doubly linked list



## insertion into and deletion from a doubly linked circular list





본 강의 자료의 그림 및 알고리즘 발췌

저자 : HOROWITZ

타이틀 : FUNDAMENTALS OF DATA STRUCTURES IN C++ 2nd Edition (2006)

공저 : SAHNI, MEHTA

 출판사 : Silicon Press