2014

Data Structure Using C++

Easy With DS

Notes For a MCA (Mgt./ Sem-III) And other Cources.

On Pune University Syllabus

Chinmay D Bhamare
AIMS Inst . Of Management ,Chalisgaon
4/18/2014

Related Topics

Data Structure using C++

Data Structures in C++

As mentioned earlier, the implementation language used in this book is C++. The reader interested in a data structures book using Java, is encouraged to consider the companion text, *Data Structures and Algorithms in Java*. C++ and Java are clearly the most widely used languages in the modern computer science curriculum. The two languages share a similar syntax and a number of important elements, such as an emphasis on object-oriented program design and strong type checking. Hence, they both incorporate important software-engineering concepts.

These two languages have advantages and disadvantages relative to each other. Java is arguably the simpler and cleaner of the two languages. It has better automatic run-time error checking, it is platform-independent, it provides automatic garbage collection, and is generally friendlier to the programmer. C++ provides a number of desirable features that are not available in Java, however. It includes stronger compile-time type checking, through the use of the **const** type modifier, and namespaces to achieve better control of the visibility of shared data. It provides efficient compile-time type polymorphism through templates, and run-time polymorphism through abstract classes and virtual functions. It includes additional data types such as enumerations and pointers. It allows explicit control over the deallocation of free-store data and destruction of class objects. For low-level code efficiency, it permits the programmer to provide hints to the compiler through inline functions and register variables. In general, C++ provides much greater control for the programmer, but it places a much greater burden of responsibility as well.

C++ was designed with the philosophy of providing a complete set of features for a procedural, object-oriented programming language, while allowing the compiler to produce very efficient code with minimal run-time overhead. Given its emphasis on efficiency, C++ continues to be a popular language for developing modern software systems. This book will help provide an understanding of the wide spectrum of skills ranging from sound algorithm and data structure design to clean and efficient implementation and coding of these designs in C++.

Defination Of OOf Data Structure

In computer science, a **data structure** is a particular way of storing and organizing data in a computer so that it can be used efficiently.[1][2]

Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers.

Data structures provide a means to manage large amounts of data efficiently, such as large databases and internet indexing services. Usually, efficient data structures are a key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design. Storing and retrieving can be carried out on data stored in both main memory and in secondary memory.

Define C++ Objects:

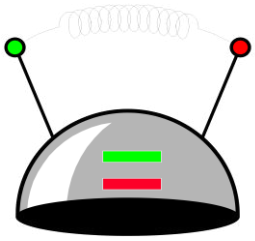
A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box:

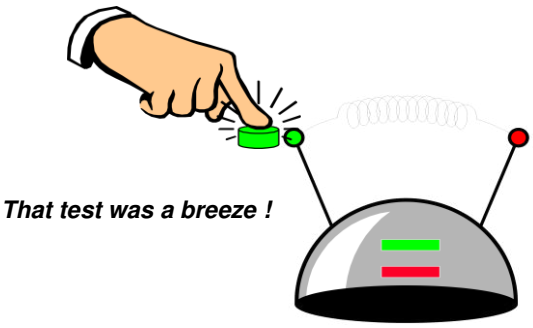
```
Box Box1;          // Declare Box1 of type Box
Box Box2;          // Declare Box2 of type Box
```

There is no real answer to the question, but we'll call it a "thinking cap".The plan is to describe a thinking cap by telling you what actions can be done to it.

Using the Object's Slots

You may put a piece of paper in each of the two slots (green and red), with a sentence written on each.You may push the green button and the thinking cap will speak the sentence from the green slot's paper.And same for the red button.





Data types

C++ pre-defines a small set of fundamental data types which include the Boolean type, several character types, several integer types, floating point types, and the special `void` type. The fundamental data types we often use are:

- `bool` (Boolean), which has one of the two values `true` or `false`
- `char` holds a character.
- `int/long int` holds an integer.
- `float/double` holds a floating point number.

(If you must know, others include things like `bool, true, false, char, wchar_t, char16_t, char32_t, int, short, long, signed, unsigned, float, double`.) See here for a more elaborate description of their value ranges.

Types that are built on other types are called compound types. For example, arrays, class/struct, pointer, reference, union, functions, and enumeration are compound types.

The C++ standard library (STL) gives us many powerful (compound) data types such as the `string` type we used above.

The `cout` object is pretty smart in the sense that it can output a variable of a fundamental data type or most STL compound types appropriately. For example,

```
#include <iostream>
using namespace std;  // generally bad habit, OK for our purposes

int main() {
    string name = "David Blaine";
    char c      = 'H';
```

```
    int i       = 12345;
    bool smart  = true;
    double avg  = 3.5;
    cout << "I am " << name << endl
        << "smart = " << smart << endl
        << "c = " << c << endl
        << "i = " << i << endl
        << "avg = " << avg << endl
        << endl;

    return 0;
}
```

Derived data types are those that are defined in terms of other data types, called base types. Derived types may have attributes, and may have element or mixed content. Instances of derived types can contain any well-formed XML that is valid according to their data type definition. They may be built-in or user-derived. Base types can be primitive types or derived data types. Derived data types are created using extension and restriction facets. They can be built-in or user-derived data types.

New types may be derived from either a primitive type or another derived type. For example, integers are a subset of real numbers. Therefore, the XML schema **integer**type is derived from the **decimal** number type, which is its base type. You can also derive an even more restricted type of integer by using the **minInclusive** and**maxInclusive** elements. This is an example of a simple type definition.

Example

The following example shows a **simpleType** element that defines a derived data type, **integer**, that is restricted to negative values.

XML

```
<simpleType name="negativeInteger">
<restriction base="xsi:integer">
<minInclusive value="-100" />
<maxInclusive value="-1" />
</restriction>
</simpleType>
```

As with the primitive types, the World Wide Web Consortium (W3C) has defined a set of built-in derived data types. These types are part of the W3C XML Schemas specification.

DATA STRUCTURE:

An implementation of abstract data type is data structure i.e. a mathematical or logical model of a particular organization of data is called data structure.

Thus, a data structure is the portion of memory allotted for a model, in which the required data can be arranged in a proper fashion.

Types:-

A data structure can be broadly classified into (i) Primitive data structure

(ii) Non-primitive data structure

(i) Primitive data structure

The data structures, typically those data structure that are directly operated upon by machine level instructions i.e. the fundamental data types such as int, float, double incase of 'c' are known as primitive data structures.

(ii) Non-primitive data structure

The data structures, which are not primitive are called non-primitive data structures.

There are two types of-primitive data structures.

(a) *Linear Data Structures:-*

A list, which shows the relationship of adjacency between elements, is said to be linear data structure. The most, simplest linear data structure is a 1-D array, but because of its deficiency, list is frequently used for different kinds of data.

(b) *Non-linear data structure:-*

A list, which doesn't show the relationship of adjacency between elements, is said to be non-linear data structure.

Linear Data Structure:

A list is an ordered list, which consists of different data items connected by means of a link or pointer. This type of list is also called a linked list. A linked list may be a single list or double linked list.

- Single linked list: - A single linked list is used to traverse among the nodes in one direction.
- Double linked list: - A double linked list is used to traverse among the nodes in both the directions.

A linked list is normally used to represent any data used in word-processing applications, also applied in different DBMS packages.
A list has two subsets. They are: -

- Stack: - It is also called as last-in-first-out (LIFO) system. It is a linear list in which insertion and deletion take place only at one end. It is used to evaluate different expressions.
- Queue: - It is also called as first-in-first-out (FIFO) system. It is a linear list in which insertion takes place at once end and deletion takes place at other end. It is generally used to schedule a job in operating systems and networks.

Non-linear data structure:-

The frequently used non-linear data structures are

(a) Trees : - It maintains hierarchical relationship between various elements
(b) Graphs : - It maintains random relationship or point-to-point relationship between various elements.

OPERATION ON DATA STRUCTURES: -

The four major operations performed on data structures are:

(i) Insertion : - Insertion means adding new details or new node into the data structure.
(ii) Deletion : - Deletion means removing a node from the data structure.
(iii) Traversal : - Traversing means accessing each node exactly once so that the nodes of a data structure can be processed. Traversing is also called as visiting.
(iv) Searching : - Searching means finding the location of node for a given key value.

Apart from the four operations mentioned above, there are two more operations occasionally

performed on data structures. They are:

(a) Sorting : - Sorting means arranging the data in a particular order.
(b) Merging : - Merging means joining two lists.

REPRESENATION OF DATA STRUCTURES:-

Any data structure can be represented in two ways. They are: -

(i) Sequential representation
(ii) Linked representation

(i) *Sequential representation*: - A sequential representation maintains the data in continuous memory locations which takes less time to retrieve the data but leads to time complexity during insertion and deletion operations. Because of sequential nature, the elements of the list must be freed, when we want to insert a new element or new data at a particular position of the list. To acquire free space in the list, one must shift the data of the list towards the right side from the position where the data has to be inserted. Thus, the time taken by CPU to shift the data will be much higher than the insertion operation and will lead to complexity in the algorithm. Similarly, while deleting an item from the list, one must shift the data items towards the left side of the list, which may waste CPU time.

Drawback of Sequential representation: -

The major drawback of sequential representation is taking much time for insertion and deletion operations unnecessarily and increasing the complexity of algorithm.

(ii) *Linked Representation:* - Linked representation maintains the list by means of a link between the adjacent elements which need not be stored in continuous memory locations. During insertion and deletion operations, links will be created or removed between which takes less time when compared to the corresponding operations of sequential representation.

Because of the advantages mentioned above, generally, linked representation is preferred for any data structure.

Algorithm Analysis:

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria.

1. Input
2. Output
3. Definiteness
4. Finiteness
5. Effectiveness

The criteria 1 & 2 require that an algorithm produces one or more outputs & have zero or more input. According to criteria 3, each operation must be definite such that it must be perfectly clear what should be done. According to the 4th criteria algorithm should terminate after a finite no. of operations. According to 5th criteria, every instruction must be very basic so that it can be carried out by a person using only pencil & paper.

There may be many algorithms devised for an application and we must analyse and validate the algorithms to judge the suitable one.

To judge an algorithm the most important factors is to have a direct relationship to the performance of the algorithm. These have to do with their computing time & storage requirements (referred as Time complexity & Space complexity).

Space Complexity:

The space complexity of an algorithm is the amount of memory it needs to run.

Time Complexity:

The time taken by a program is the sum of the compiled time & the run time. The time complexity of an algorithm is given by the number of steps taken by the algorithm to compute the function it was written for.

[NOTE : example on how to calculate computing time will be given later]

Storage Structure for Arrays

Array is set of homogenous data items represented in contiguous memory locations using a common name and sequence of indices starting from 0. Array is a simplest data structure that makes use of computed address to locate its elements. An array size is fixed and therefore requires a fixed number of memory locations.

Suppose A is an array of n elements and the starting address is given then the location and element I will be

$LOC(A_i)$ = Base address of A + (i – 1) * W

Where W is the width of each element.

A multidimensional array can be represented by an equivalent one-dimensional array. A two dimensional array consisting of number of rows and columns is a combination of more than 1 one-dimensional array. A 2 dimensional array is referred in two different ways. Considering row as major order or column as major order any array may be used to refer the elements.

If we consider the row as major order then the elements are referred row by row whose addressing function may be

$LOC(A_{rc})$ = Base address of A + [(r-1) * N + (c-1)] * W

Where r and c are subscripts. N is number of columns per row. W is the width of each element.

If we consider the column as major order then the elements are referred column by column.

Sparse Matrices

Matrices with relatively high proportion of zero or null entries are called sparse matrices.

When matrices are sparse, then much space and computing time could e saved if the non-zero

entries were stored explicitly i.e. ignoring the zero entries the processing time and space can be minimized in sparse matrices.

$$\begin{pmatrix} 0 & 0 & 0 & 24 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{matrix} 0 & 0 & 0 & 0 & 9 & 0 & 0 \\ 0 & 0 & 0 & 0 & 18 & 0 & 0 \\ 0 & 0 & 0 & 0 & 8 & 0 & 0 \end{matrix}$$

In the above matrix we have 6 rows and 7 columns. There are 5 nonzero entries out of 42 entries. It requires an alternate form to represent the matrix without considering the null entries.

The alternate data structure that we consider to represent a sparse matrix is a triplet. The triplet is a two dimensional array having t+1 rows and 3 columns. Where, t is total number of nonzero entries.

The first row of the triplet contains number of rows, columns and nonzero entries available in the matrix in its 1st, 2nd and 3rd column respectively. Second row onwards it contains the row subscript, column subscript and the value of the nonzero entry in its 1st, 2nd and 3rd column respectively.

Let us represent the above matrix in the following triplet of 6 rows and 3 columns

6	7	5
1	4	24
2	6	5
4	5	9
5	5	18
6	5	8

The above triplet contains only non-zero details by reducing the space for null entries.

[Follow the algorithms taught in class]

Stacks

A stack is a linear data structure in which an element may be inserted or deleted only at one end called the top end of the stack i.e. the elements are removed from a stack in the reverse order of that in which they were inserted into the stack.

A stack follows the principle of last-in-first-out (LIFO) system. According to the stack terminology, PUSH and POP are two terms used for insert and delete operations.

Representation of Stacks

A stack may be represented by means of a one way list or a linear array. Unless, otherwise stated, each of the stacks will be maintained by a linear array STACK, A variable TOP contains the location of the top element of the stack. A variable N gives the maximum number elements that can be held by the stack. The condition where TOP is NULL, indicate that the stack is empty. The condition where TOP is N, will indicate that the stack is full.

[Follow the push and pop algorithms discussed in class]

Application of Stacks

There are two important applications of stacks.

a) Recursion

b) Arithmetic Expression

Recursion

Recursion is and important facility in many programming languages. There are many problems whose algorithmic description is best described in a recursive manner.

A function is called recursive if the function definition refers to itself or does refers to another function which in turn refers back to the same function. In-order for the definition not to be circular, it must have the following properties:

(i) There must be certain arguments called base values, for which the function does not refer to itself.

(ii) Each time the function does refer to itself, the argument of the function must be closer to a base value.

A recursive function with those two properties is said to be well defined.

Let us consider the factorial of a number and its algorithm described recursively:

We know that N! = N * (N-1)!

(N-1)! = N-1 * (N-2)! and so on up to 1.

FACT(N)

1. if N=1
   return 1

2. else
   return N * FACT(N-1)

3. end

Let N be 5.

Then according to the definition FACT(5) will call FACT(4), FACT(4) will call FACT(3), FACT(3) will call FACT(2), FACT(2) will call FACT(1). Then the execution will return back by finishing the execution of FACT(1), then FACT(2) and so on up to FACT(5) as described below.

1) 5! = 5 * 4!
2) 4! = 4 * 3!
3) 3! = 3 * 2!
4) 2! = 2 * 1!
5) 1! = 1
6) 2! = 2 * 1 = 2
7) 3! = 3 * 2 = 6
8) 4! = 4 * 6 = 24
9) 5! = 5 * 24 = 120

From above example it is clear that every sub function contain parameters and local variables. The parameters are the arguments which receive values from objects in the calling program and which transmit values back to the calling program. The sub-function must also keep track of the return address in the calling program. This return address is essential since control must be transferred back to its proper place in the calling program. After completion of the sub-function when the control is transferred back to its calling program, the local values and returning address is no longer needed. Suppose our sub-program is a recursive one, when it call itself, then current values must be saved, since they will be used again when the program is reactivated.

Thus, in recursive process a data structure is required to handle the data of ongoing called function and the function which is called at last must be processed first. i.e the data accessed last must be processed fist i.e Last in first out principle. So, a stack may be suitable data structure that follows LIFO to implement recursion.

Arithmetic Expression

This section deals with the mechanical evaluation or compilation of infix expression. The stack is find to be more efficient to evaluate an infix arithmetical expression by first converting to a suffix or postfix expression and then evaluating the suffix expression. This approach will eliminate the repeated scanning of an infix expressions in order to obtain its value.

A normal arithmetic expression is normally called as infix expression. E.g A+B

A Polish mathematician found a way to represent the same expression called polish notation or prefix expression by keeping operators as prefix. E.g +AB

We use the reverse way of the above expression for our evaluation. The representation is called

Reverse Polish Notation (RPN) or postfix expression. E.g. AB+

The arithmetic expression evaluation is performed in two phases, they are

- Conversion of infix to postfix expression
- Evaluation of postfix expression

[Follow the algorithms and examples discussed in class]

Queue

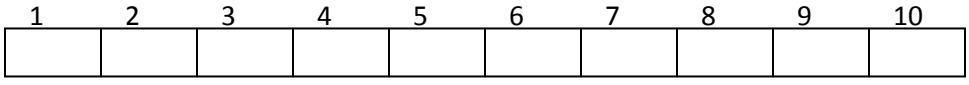
Queue is a linear data structure in which insertion can take place at only one end called rear end and deletion can take place at other end called top end. The front and rear are two terms used to represent the two ends of the list when it is implemented as queue. Queue is also called First In First Out (FIFO) system since the first element in queue will be the first element out of the queue.

Like stacks, queues may be represented in various ways, usually by means of one way list or linear arrays. Generally, they are maintained in linear array QUEUE. Two pointers FRONT and REAR are used to represent front and last element respectively. N may be the size of the linear array. The condition when FRONT is NULL indicate that the queue is empty. The condition when REAR is N indicated overflow.

[Follow the algorithms for insert and delete discussed in class]

Circular Queue

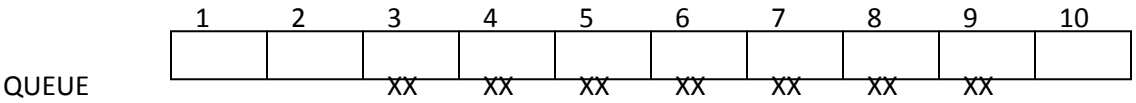
The linear arrangement of the queue always considers the elements in forward direction. In the above two algorithms, we had seen that, the pointers front and rear are always incremented as and when we delete or insert element respectively. Suppose in a queue of 10 elements front points to 4^{th} element and rear points to 8^{th} element as follows.



QUEUE XX XX XX XX XX

FRONT REAR

When we insert two more elements then the array will become



FRONT REAR

Later, when we try to insert some elements, then according to the logic when REAR is N then it encounters an overflow situation. But there are some elements are left blank at the beginning part of the array. To utilize those left over spaces more efficiently, a circular fashion is implemented in queue representation. The circular fashion of queue reassigns the rear pointer with 1 if it reaches N and beginning elements are free and the process is continued for deletion also. Such queues are called Circular Queue.

[Follow the CQINSERT and CDELETE algorithm]

Types of QUEUE

There are two types of Queue

- Priority Queue
- Double Ended Queue

Priority Queue

A priority queue is a collection of elements such that each element has been assigned a priority value such that the order in which elements are deleted and processed comes from the following rules

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were added to the queue.

There are various ways of maintaining a priority queue in memory. One is using one way list. The sequential representation is never preferred for priority queue. We use linked Queue for priority Queue.

Double Ended Queue

A Double Ended Queue is in short called as Deque (pronounced as Deck or dequeue). A deque is a linear queue in which insertion and deletion can take place at either ends but not in the middle.

There are two types of Deque.

1. Input restricted Deque
2. Output restricted Deque

A Deque which allows insertion at only at one end of the list but allows deletion at both the ends of the list is called Input restricted Deque.

A Deque which allows deletion at only at one end of the list but allows insertion at both the ends of the list is called Output restricted Deque.

Implementation of Data Structure

Heaptree and result list are stored together in one array. The array has as many elements as there are keys to sort. No additional storage space for pointers or intermediate results is needed. The beginning of the array is used by the heaptree, the remaining end by the result list (figure 1).

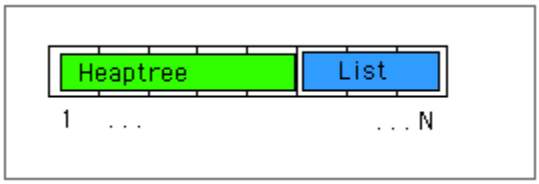


Figure 1: Heaptree and result list stored in one array

Providing Input Data

A program that wants to sort data with the heapsort algorithm provides it as an array. The array appears to the algorithm as a heaptree containing all keys. In the notation of figure 1 the green heaptree occupies the whole array. That is why the heaptree needs not be built but exists instantly, complete and completely filled.

Running the algorithm

The function heapsort performs the complete algorithm. Within the algorithm the function move-max moves keys from the heaptree to the result list. This removes one key from the heaptree which makes room in the array at the border between heaptree and result list. The result list grows using this free node and stores the new element there.

Providing Output Data

After the algorithm run the result list occupies the complete array. There is no need to copy this list, because the array can be used directly by the program.

Mapping Heaptree Nodes to Array Nodes

The heaptree root is stored at index 1 in the array.

For a node stored at index k in the array, its left child is stored at index 2*k and its right child at index 2*k+1 (figure 2). It follows that the parent node is at index (k div 2).

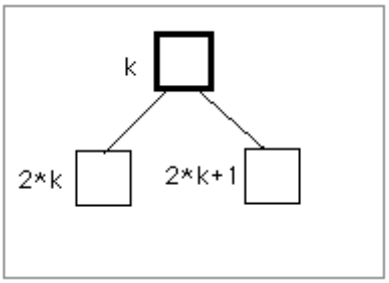


Figure 2: Mapping child nodes to indices

Figure 3 shows the array indices of the heaptree nodes for the heaptree used in the courseware.

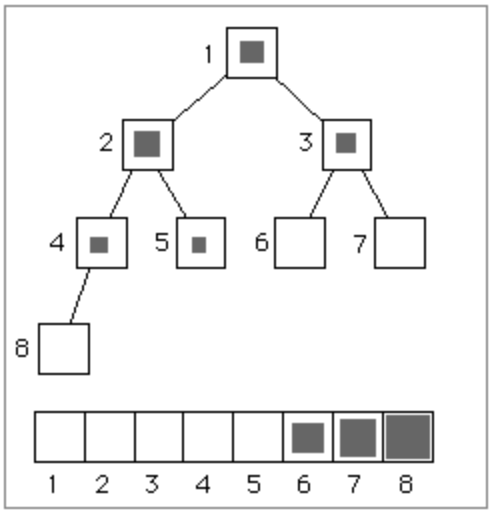


Figure 3: Array indices of heaptree nodes

Arrays

Array is a collection mainly using similar data types that are stored into a common variable, forming (at least conceptually that may even be replicated into the memory hardware) a linear data structure.

An array is a particular method of storing **elements** of indexed data. Elements of data are logically stored sequentially in blocks within the array. Each element is referenced by an**index**, or subscripts.

The index is usually a number used to address an element in the array. For example, if you were storing information about each day in August, you would create an array with an index capable of addressing 31 values -- one for each day of the month. Indexing rules are language dependent, however most languages use either 0 or 1 as the first element of an array.

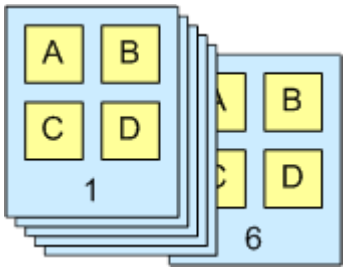
The concept of an array can be daunting to the uninitiated, but it is really quite simple. Think of a

notebook with pages numbered 1 through 12. Each page may or may not contain information on it. The notebook is an *array* of pages. Each page is an *element* of the array 'notebook'. Programmatically, you would retrieve information from a page by referring to its number or *subscript*, i.e., notebook(4) would refer to the contents of page 4 of the array notebook.

The notebook (array) contains 12 pages (elements)

Arrays can also be multidimensional - instead of accessing an element of a one-dimensional list, elements are accessed by two or more indices, as from a matrix or tensor.

Multidimensional arrays are as simple as our notebook example above. To envision a multidimensional array, think of a calendar. Each page of the calendar, 1 through 12, is an element, representing a month, which contains approximately 30 elements, which represent days. Each day may or may not have information in it. Programmatically then, calendar(4,15) would refer to the 4th month, 15th day. Thus we have a two-dimensional array. To envision a three-dimensional array, break each day up into 24 hours. Now calendar(4,15,9) would refer to 4th month, 15th day, 9th hour.



A simple 6 element by 4 element array

Array<Element> Operations

`make-array(integer` *`n`*`): Array`

Create an array of elements indexed from 0 to $n-1$, inclusive. The number of elements in the array, also known as the size of the array, is *n*.

`get-value-at(Array` *`a`*`, integer` *`index`*`): Element`

Returns the value of the element at the given *index*. The value of *index* must be in bounds: *0 <= index <= (n - 1)*. This operation is also known as**subscripting**.

`set-value-at(Array` *`a`*`, integer` *`index`*`, Element` *`new-value`*`)`

Sets the element of the array at the given *index* to be equal to *new-value*.

http://en.wikibooks.org/wiki/Data_Structures/Arrays

Type

The array index needs to be of some type. Usually, the standard integer type of that language is used, but there are also languages such as Ada and Pascal which allow any discrete type as an array index. Scripting languages often allow any type as an index (associative array).

Bounds

The array index consists of a range of values with a lower bound and an upper bound.

In some programming languages only the upper bound can be chosen while the lower bound is fixed to be either 0 (C, C++,C#, Java) or 1 (FORTRAN 66, R).

In other programming languages (Ada, PL/I, Pascal) both the upper and lower bound can be freely chosen (even negative).

Bounds check

The third aspect of an array index is the check for valid ranges and what happens when an invalid index is accessed. This is a very important point since the majority of computer worms and computer viruses attack by using invalid array bounds.

There are three options open:

1. Most languages (Ada, PL/I, Pascal, Java, C#) will check the bounds and raise some error condition when an element is accessed which does not exist.
2. A few languages (C, C++) will not check the bounds and return or set some arbitrary value when an element outside the valid range is accessed.
3. Scripting languages often automatically expand the array when data is written to an index which was not valid until then.

Declaring Array Types

The declaration of array type depends on how many features the array in a particular language has.

The easiest declaration is when the language has a fixed lower bound and fixed index type. If you need an array to store the monthly income you could declare in C

```
typedef double Income[12];
```

This gives you an array with in the range of 0 to 11. For a full description of arrays in C see C Programming/Arrays.

If you use a language where you can choose both the lower bound as well as the index type, the declaration is -- of course -- more complex. Here are two examples in Ada:

```
type Month is range 1 .. 12;
type Income is array(Month) of Float;
```

or shorter:

```
type Income is array(1 .. 12) of Float;
```

For a full description of arrays in Ada see Ada Programming/Types/array.

Array Access

We generally write arrays with a name, followed by the index in some brackets, square '[]' or round '()'. For example,`august[3]` is the method used in the C programming language to refer to a particular day in the month.

Because the C language starts the index at zero, `august[3]` is the 4th element in the array. `august[0]` actually refers to the first element of this array. Starting an index at zero is natural for computers, whose internal representations of numbers begin with zero, but for humans, this unnatural numbering system can lead to problems when accessing data in an array. When fetching an element in a language with zero-based indexes, keep in mind the *true* length of an array, lest you find yourself fetching the wrong data. This is the disadvantage of programming in languages with fixed lower bounds, the programmer must always remember that "[0]" means "1st" and, when appropriate, add or subtract one from the index. Languages with variable lower bounds will take that burden off the programmer's shoulder.

We use indexes to store *related* data. If our C language array is called `august`, and we wish to store that we're going to the supermarket on the 1st, we can say, for example

```
august[0] = "Going to the shops today"
```

http://en.wikibooks.org/

Polynomial Representation Using Arrays

Create a data structure “polynomial” which has an int pointer to store the co-efficients of the elements. Also has an integer variable to store the maximum degree of the polynomial (order of the polynomial). The class should have the following functionality: (use one dimensional array)

• Create(int) – which takes the maximum degree of the polynomial as parameter and create a dynamic array accordingly.

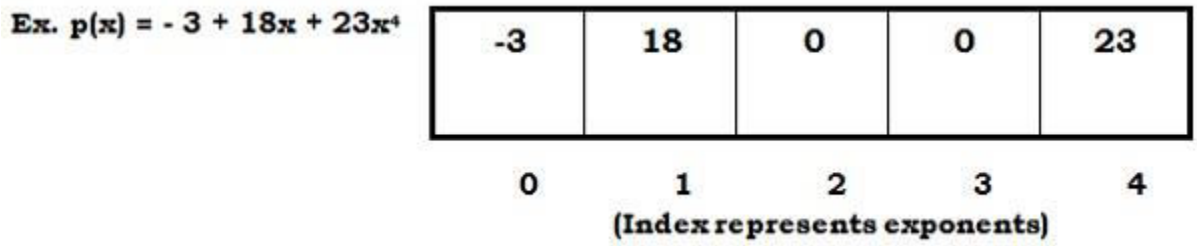
• Accept() – which accepts the co-efficient of each degree and stores it in the array.

• Display() – Which displays the polynomial in the form of equation.

• Polynomial operator +(polynomial) – Overload the operator + to perform addition of two polynomials and return the resultant polynomial.

• Polynomial operator *(polynomial) – overload the operator * to perform multiplication of two polynomials and return the resultant polynomial.

Note: Polynomials have co-efficients and exponents. Array data structure has been used to implement addition and multiplication of polynomials. In this, Index of the array represents exponents. Co-efficients are stored in the array, in the respective position of the exponent (index of the array).



Solution

```
#include <iostream.h>
#include <conio.h>
class poly
{ int *coeff;
int dmax;
public:
void create(int);
void accept();
void display();
poly operator +(poly);
poly operator *(poly);
};
void poly::create( int m)
```

```
{ dmax=m;
coeff=new int[dmax+1];
for(int i=0;i<=dmax;i++)
coeff[i]=0;
}


void poly::accept()
{
for(int i=0;i<=dmax;i++)
{
cout<<"Enter the co-efficient at degree "<<i<<":";
cin>>coeff[i];
}
}

void poly::display()
{
cout<<endl<<"The polynomial is :";
for(int i=0;i<=dmax;i++)
{
if(coeff[i]!=0)
cout<< coeff[i] << "X^"<< i << " + ";
}
cout<<endl;
}

poly poly::operator *(poly p)
{
int s=dmax+p.dmax;
poly temp;
temp.create(s);
for(int i=0;i<=dmax;i++)
```

```
{
for(int j=0;j<=p.dmax;j++)
temp.coeff[i+j]+=(coeff[i] * p.coeff[j]);
}
return temp;
}

poly poly::operator +(poly p)
{
poly temp;
int small, large, flag;
if(dmax>p.dmax)
{
large=dmax;
small=p.dmax;
flag=1;
}
else
{
large=p.dmax;
small=dmax;
flag=0;
}
temp.create(large);
for(int i=0;i<=small;i++)
{
temp.coeff[i]=coeff[i]+p.coeff[i];
}
for(i=small+1;i<=large;i++)
{
if(flag==1)
temp.coeff[i]=coeff[i];
else
```

```
temp.coeff[i]=p.coeff[i];
}
return temp;
}

void main()
{
clrscr();
poly p1,p2,p3;
cout<<"Enter the order of your first polynomial :";
int deg;
cin>>deg;
p1.create(deg);
p1.accept();
p1.display();

cout<<"Enter the order of your second polynomial :";
cin>>deg;
p2.create(deg);
p2.accept();
p2.display();

p3=p1+p2;
cout<<"Resultant polynomial after adding two polynomials"<<endl;
p3.display();

p3=p1*p2;
cout<<"Resultant polynomial after multiplying two polynomials"<<endl;
p3.display();
getch();
```

Program Using Two Polynomial

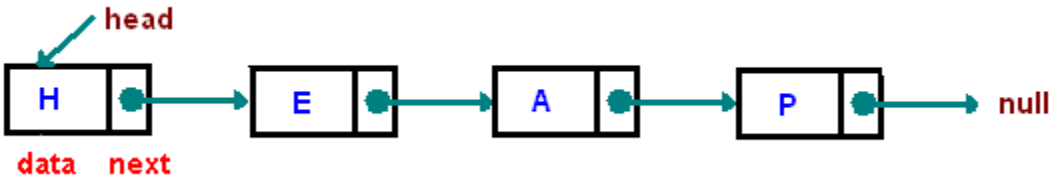
```
#include <stdio.h>

#define MAX 10

int main(){
int poly1[MAX]={0},poly2[MAX]={0},poly3[MAX]={0};
int i,deg1,deg2,deg3;
printf("nEnter degree of first polynomial?");
scanf("%d",&deg1);
printf("nEnter degree of second polynomial?");
scanf("%d",&deg2);

printf("nFor first polynomial:");
for(i=0;i<deg1;i++){
printf("nEnter Coefficient for Exponent %d> ",i);
scanf("%d",&poly1[i]);
}
printf("nFor second polynomial:");
for(i=0;i<deg2;i++){
printf("nEnter Coefficient for Exponent %d> ",i);
scanf("%d",&poly2[i]);
}
printf("nGenerating sum…!");
printf("nPress any key to continue…");

deg3 = (deg1>deg2)?deg1:deg2;

for(i=0;i<deg3;i++)
poly3[i] = poly1[i] + poly2[i];
printf("nnAddition Result:nn");

for(i=deg1-1;i>=0;i–)
printf("(%dx^%d)+",poly1[i],i);
printf("b n");

for(i=deg2-1;i>=0;i–)
printf("(%dx^%d)+",poly2[i],i);
printf("b n");

for(i=deg3-1;i>=0;i–)
printf("(%dx^%d)+",poly3[i],i);
printf("b n");

}
```

Linked Lists

Introduction

One disadvantage of using arrays to store data is that arrays are static structures and therefore cannot be easily extended or reduced to fit the data set. Arrays are also expensive to maintain new insertions and deletions. In this chapter we consider another data structure called Linked Lists that addresses some of the limitations of arrays.

A linked list is a linear data structure where each element is a separate object.



Each element (we will call it a **node**) of a list is comprising of two items - the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the **head** of the list. It should be noted that head is not a separate node, but the reference to the first node. If the list is empty then the head is a null reference.

A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list.

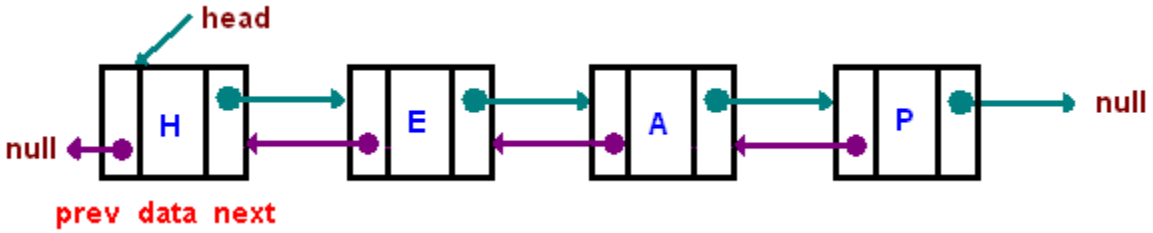
One disadvantage of a linked list against an array is that it does not allow direct access to the individual elements. If you want to access a particular item then you have to start at the head and follow the references until you get to that item.

Another disadvantage is that a linked list uses more memory compare with an array - we extra 4 bytes (on 32-bit CPU) to store a reference to the next node.

Types of Linked Lists

A **singly linked list** is described above

A **doubly linked list** is a list that has two references, one to the next node and another to previous node.



Another important type of a linked list is called a **circular linked list** where last node of the list points back to the first node (or the head) of the list.

The Node class

In Java you are allowed to define a class (say, B) inside of another class (say, A). The class A is called the outer class, and the class B is called the **inner**class. The purpose of inner classes is purely to be used internally as helper classes. Here is the LinkedList class with the inner Node class

```
private static class Node<AnyType>
{
   private AnyType data;
   private Node<AnyType> next;

   public Node(AnyType data, Node<AnyType> next)
   {
      this.data = data;
      this.next = next;
   }
}
```

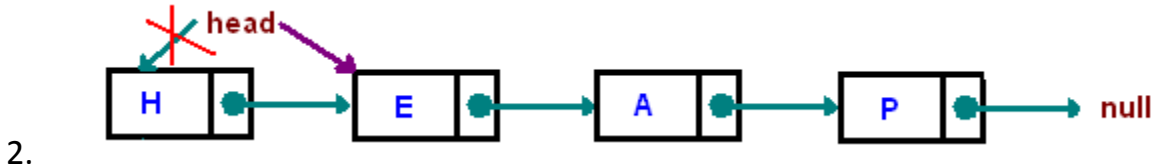
An inner class is a member of its enclosing class and has access to other members (inclusing private) of the outer class, And vise versa, the outer class can have a direct access to all members of the inner class. An inner class can be declared private, public, protected, or package private. There are two kind of inner classes: static and non-static. A static inner class cannot refer directly to instance variables or methods defined in its outer class: it can use them only through an object reference.

We implement the LinkedList class with two inner classes: static Node class and non-static LinkedListIterator class. See LinkedList.java for a complete implementation.

Examples

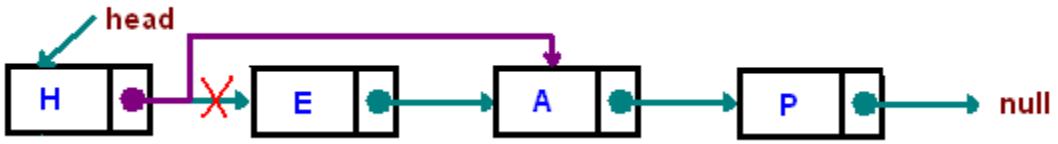
Let us assume the singly linked list above and trace down the effect of each fragment below. The list is restored to its initial state before each line executes

```
1. head = head.next;
```



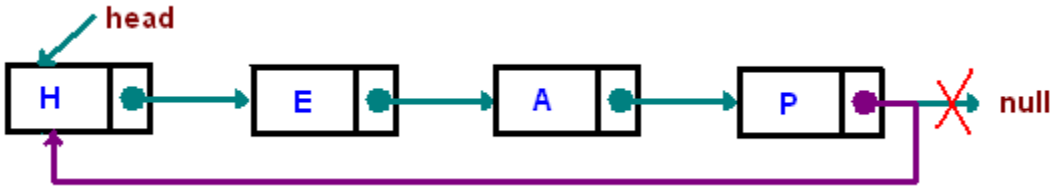
2.

```
3. head.next = head.next.next;
```



4.

```
5. head.next.next.next.next = head;
```

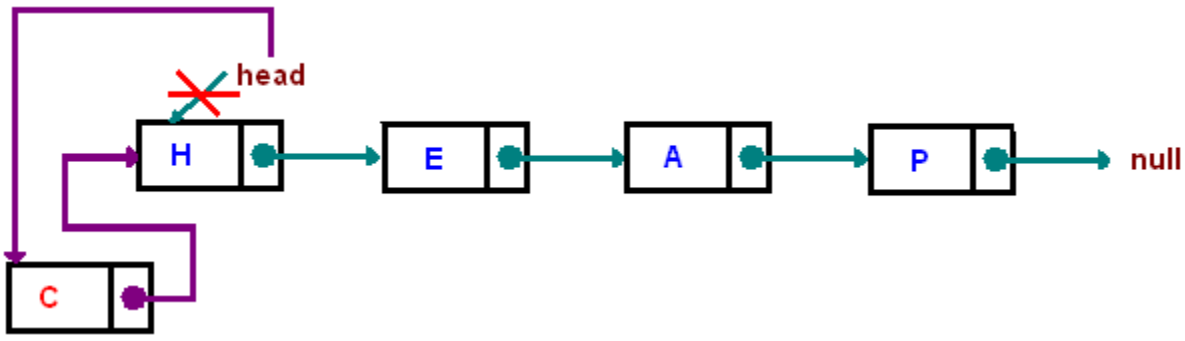


6.

Linked List Operations

addFirst

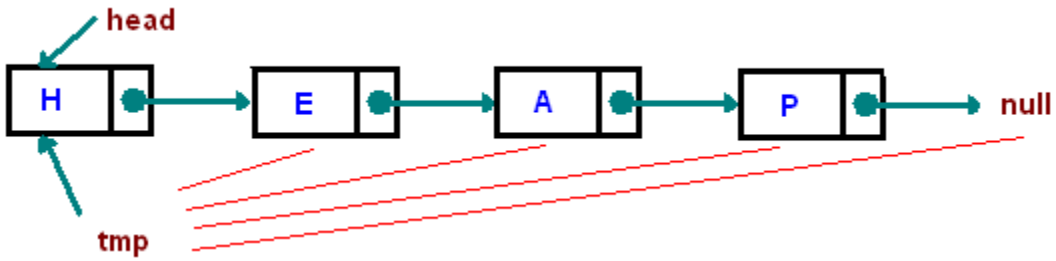
The method creates a node and prepends it at the beginning of the list.



```
public void addFirst(AnyType item)
{
   head = new Node<AnyType>(item, head);
}
```

Traversing

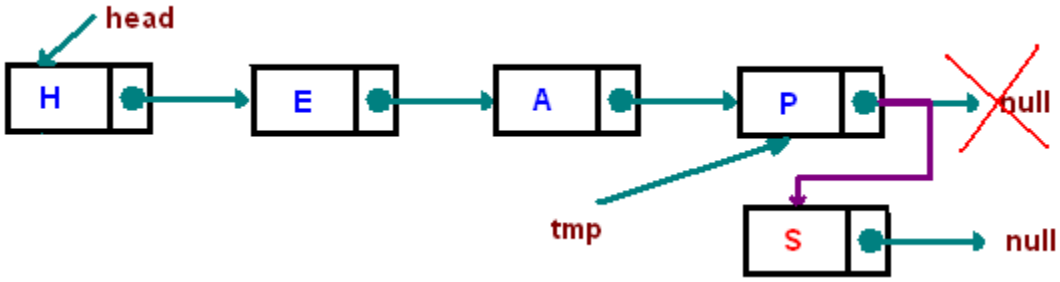
Start with the head and access each node until you reach null. Do not change the head reference.




```
Node tmp = head;

while(tmp != null) tmp = tmp.next;
```

addLast

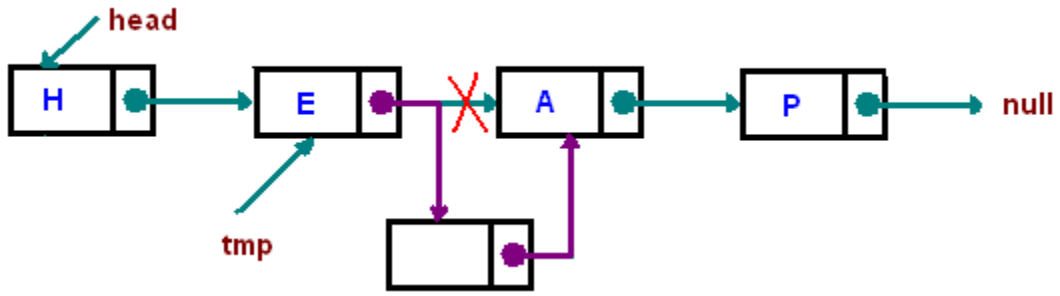
The method appends the node to the end of the list. This requires traversing, but make sure you stop at the last node




```
public void addLast(AnyType item)
{
   if(head == null) addFirst(item);
   else
   {
      Node<AnyType> tmp = head;
      while(tmp.next != null) tmp = tmp.next;

      tmp.next = new Node<AnyType>(item, null);
   }
}
```

Inserting "after"

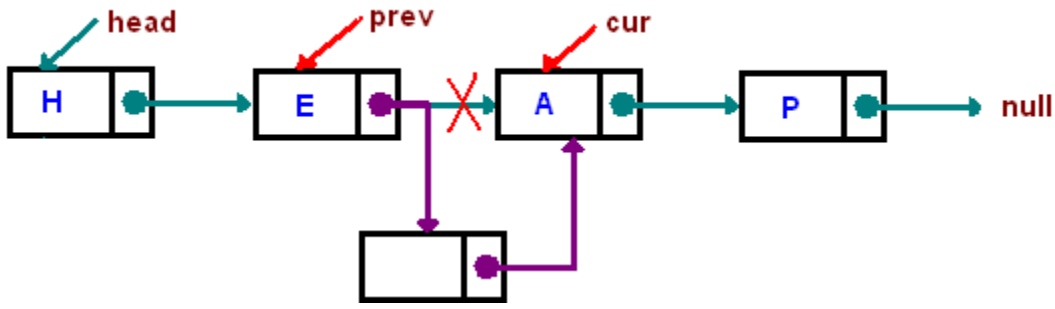
Find a node containing "key" and insert a new node after it. In the picture below, we insert a new node after "e":




```
public void insertAfter(AnyType key, AnyType toInsert)
{
   Node<AnyType> tmp = head;
   while(tmp != null && !tmp.data.equals(key)) tmp = tmp.next;

   if(tmp != null)
      tmp.next = new Node<AnyType>(toInsert, tmp.next);
}
```

Inserting "before"

Find a node containing "key" and insert a new node before that node. In the picture below, we insert a new node before "a":



For the sake of convenience, we maintain two references `prev` and `cur`. When we move along the list we shift these two references, keeping `prev` one step before `cur`. We continue until `cur` reaches the node before which we need to make an insertion. If `cur` reaches null, we don't insert, otherwise we insert a new node between `prev` and `cur`.

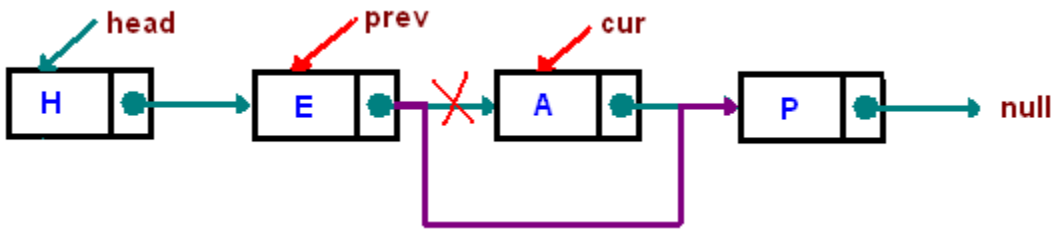
Examine this implementation

```
public void insertBefore(AnyType key, AnyType toInsert)
{
```

```
   if(head == null) return null;
   if(head.data.equals(key))
   {
      addFirst(toInsert);
      return;
   }

   Node<AnyType> prev = null;
   Node<AnyType> cur = head;

   while(cur != null && !cur.data.equals(key))
   {
      prev = cur;
      cur = cur.next;
   }
   //insert between cur and prev
   if(cur != null) prev.next = new Node<AnyType>(toInsert, cur);
}
```

Deletion

Find a node containing "key" and delete it. In the picture below we delete a node containing "A"



The algorithm is similar to insert "before" algorithm. It is convinient to use two references `prev` and `cur`. When we move along the list we shift these two references, keeping `prev` one step before `cur`. We continue until `cur` reaches the node which we need to delete. There are three exceptional cases, we need to take care of:

1. list is empty
2. delete the head node
3. node is not in the list

```
public void remove(AnyType key)
{
   if(head == null) throw new RuntimeException("cannot delete");

   if( head.data.equals(key) )
   {
      head = head.next;
      return;
   }
```

```
   Node<AnyType> cur  = head;
   Node<AnyType> prev = null;

   while(cur != null && !cur.data.equals(key) )
   {
      prev = cur;
      cur = cur.next;
   }

   if(cur == null) throw new RuntimeException("cannot delete");

   //delete cur node
   prev.next = cur.next;
}
```

Iterator

The whole idea of the iterator is to provide an access to a private aggregated data and at the same moment hiding the underlying representation. An iterator is Java is an object, and therefore it's implementation requires creating a class that implements the *Iterator* interface. Usually such class is implemented as a private inner class. The *Iterator* interface contains the following methods:

- AnyType next() - returns the next element in the container
- boolean hasNext() - checks if there is a next element
- void remove() - (optional operation).removes the element returned by next()

In this section we implement the Iterator in the LinkedList class. First of all we add a new method to the LinkedList class:

```
public Iterator<AnyType> iterator()
{
   return new LinkedListIterator();
}
```

Here `LinkedListIterator` is a private class inside the LinkedList class

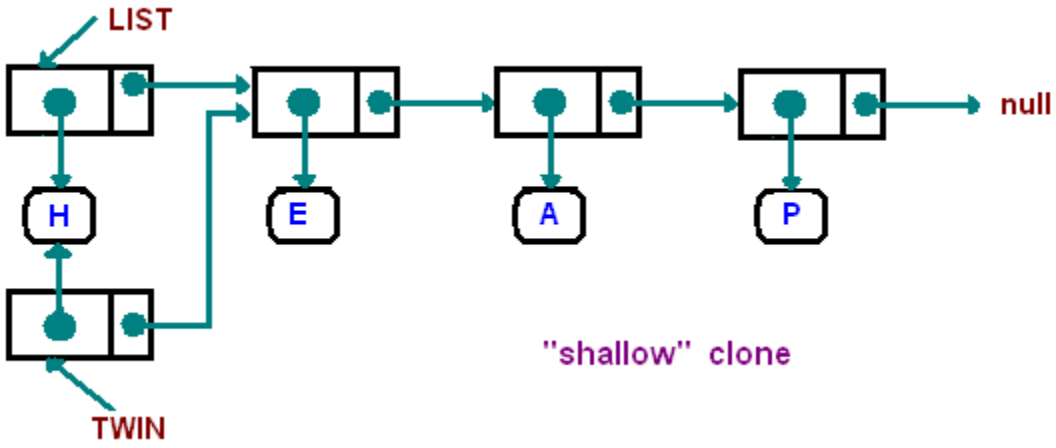
```
private class LinkedListIterator implements Iterator<AnyType>
{
   private Node<AnyType> nextNode;

   public LinkedListIterator()
   {
      nextNode = head;
   }
   ...
}
```

The `LinkedListIterator` class must provide implementations for `next()` and `hasNext()` methods. Here is the `next()` method:

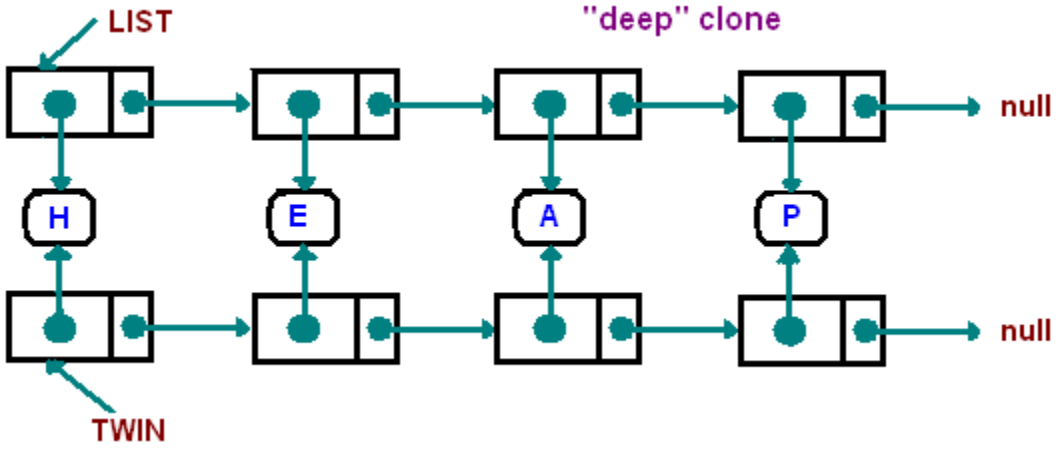
```
public AnyType next()
{
   if(!hasNext()) throw new NoSuchElementException();
   AnyType res = nextNode.data;
   nextNode = nextNode.next;
   return res;
}
```

Cloning

Like for any other objects, we need to learn how to clone linked lists. If we simply use the clone() method from the Object class, we will get the following structure called a "shallow" copy:



The Object's clone() will create a copy of the first node, and share the rest. This is not exactly what we mean by "a copy of the object". What we actually want is a copy represented by the picture below



Since out data is immutable it's ok to have data shared between two linked lists. There are a few ideas to implement linked list copying. The simplest one is to traverse the original list and copy each node by using the addFirst() method. When this is finished, you will have a new list in the reverse order. Finally, we will have to reverse the list:

```
public Object copy()
{
   LinkedList<AnyType> twin = new LinkedList<AnyType>();
   Node<AnyType> tmp = head;
   while(tmp != null)
   {
      twin.addFirst( tmp.data );
      tmp = tmp.next;
   }

   return twin.reverse();
}
```

A better way involves using a tail reference for the new list, adding each new node after the last node.

```
public LinkedList<AnyType> copy3()
{
   if(head==null) return null;
   LinkedList<AnyType> twin = new LinkedList<AnyType>();
   Node tmp = head;
   twin.head = new Node<AnyType>(head.data, null);
   Node tmpTwin = twin.head;

   while(tmp.next != null)
   {
      tmp = tmp.next;
      tmpTwin.next = new Node<AnyType>(tmp.data, null);
      tmpTwin = tmpTwin.next;
   }

   return twin;
}
```

Polynomial Algebra

The biggest integer that we can store in a variable of the type int is $2^{31} - 1$ on 32-but CPU. You can easily verify this by the following operations:

```
int prod=1;
for(int i = 1; i <=; 31; i ++)
         prod *= 2;
System.out.println(prod);
```

This code doesn't produce an error, it produces a result! The printed value is a *negative* integer `-2147483648 = -2`^{31}. If the value becomes too large, Java saves only the low order 32 (or 64 for longs) bits and throws the rest away.

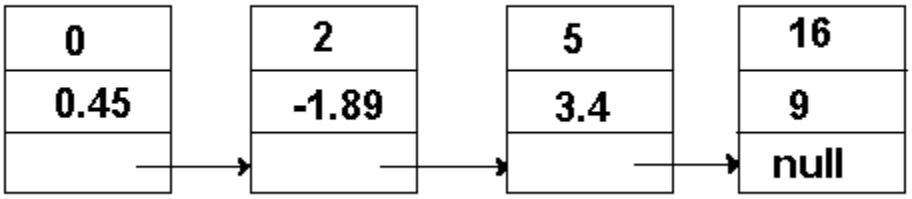
In real life applications we need to deal with integers that are larger than 64 bits (the size of a long). To manipulate with such big numbers, we will be using a linked list data structure. First we observe that each integer can be expressed in the decimal system of notation.

$$937 = 9*10^2 + 3*10^1 + 7*10^0$$
$$2011 = 2*10^3 + 0*10^2 + 1*10^1 + 1*10^0$$

Now, if we replace a decimal base 10 by a character, say 'x', we obtain a univariate polynomial, such as

$$0.45 - 1.89\ x^2 + 3.4\ x^5 + 9\ x^{16}$$

We will write an application that manipulates polynomials in one variable with real coefficients.Among many operations on polynomials, we implement addition, multiplication, differentiation and evaluation. A polynomial willbe represented as a linked list, where each node has an integer degree, a double coefficient and a reference to the next term. The final node will have a null reference to indicate the end of the list. Here is a linked link representation for the above polynomial:



The linked list data structures that we implemented in Chapter 5 are characterized by a *linear* (line-like) relationship between the elements: Each element (except the first one) has a unique predecessor, and each element (except the last one) has a unique successor. Using linear linked lists does present a problem: Given a pointer to a node anywhere in the list, we can access all the nodes that follow but none of the nodes that precede it. With a singly linked linear list structure (a list whose pointers all point in the same direction), we must always have a pointer to the beginning of the list to be able to access all the nodes in the list.

In addition, the data we want to add to a sorted list may already be in order. Sometimes people manually sort raw data before turning it over to a data entry clerk. Likewise, data produced by other programs are often in some order. Given a Sorted List ADT and sorted input data, we

always insert new items at the end of the list. It is ironic that the work done manually to order the data now results in maximum insertion times.

We can, however, change the linear list slightly, making the pointer in the next member of the last node point back to the first node instead of containing NULL (Figure 6.1). Now our list becomes a circular linked list rather than a linear linked list. We can start at any node in the list and traverse the entire list. If we let our external pointer point to the last item in the list rather than the first, we have direct access to both the first and the last elements in the list (Figure 6.2). listData->info references the item in the last node, and listData-> next->info references the item in the first node. We mentioned this type of list structure in Chapter 5, when we discussed circular linked queues.

Figure 6.1: A circular linked list

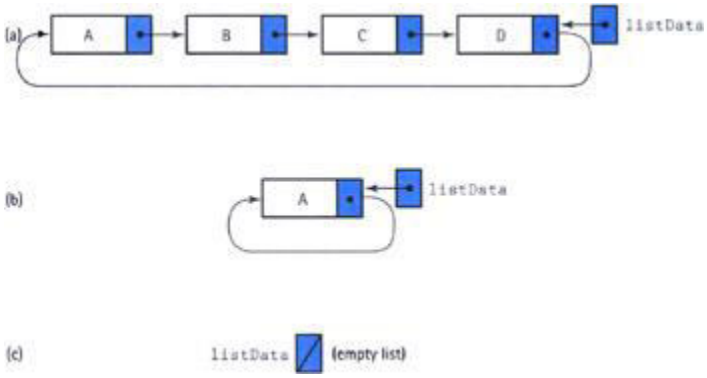


Figure 6.2: Circular linked lists with the external pointer pointing to the rear element

Circular linked list A list in which every node has a successor; the "last" element is succeeded by the "first" element

We do not need to change any of the declarations in the class SortedType to make the list circular, rather than linear. After all, the members in the nodes are the same; only the value of the next member of the last node has changed. How does the circular nature of the list alter the implementations of the list operations? Because an empty circular list has a NULL pointer, the IsEmpty operation does not change at all. However, using a circular list requires an obvious change in the algorithms that traverse the list. We no longer stop when the traversing pointer becomes NULL. Indeed, unless the list is empty, the pointer never becomes NULL. Instead, we must look for the external pointer itself as a stop sign. Let's examine these changes in the Sorted List ADT.

Finding a List Item

The RetrieveItem, InsertItem, and DeleteItem operations all require a search of the list. Rather than rewriting each of these with minor variations, let's write a generalFindItem routine that

takes item as a parameter and returns location, predLoc, and found.
InsertItem and DeleteItem need the location of the predecessor node (predLoc); RetrieveItem can just ignore it.

In the linear list implementation, we searched the list using a pair of pointers, location and predLoc. (Remember the inchworm?) We modify this approach slightly for the circular list. In the linear list version, we initialized location to point to the first node in the list and set predLoc to NULL (Figure 6.3a). For the circular list search, we initializelocation to point to the first node and predLoc to point to its "predecessor"-the last node in the list (Figure 6.3b).

The search loop executes until (1) a key greater than or equal to the item's key is encountered, or (2) we reach the "end" of the list. In a linear list, the end of the list is detected when location equals NULL. Because the external pointer to the list points to the last element, we know we have processed all of the items and not found a match when location points to the first element again: location = listData->next. Because it makes no sense to search an empty list, let's make it a precondition that the list is not empty. Because we now have overloaded the relational operators "<" and "==", we use them in our algorithm.

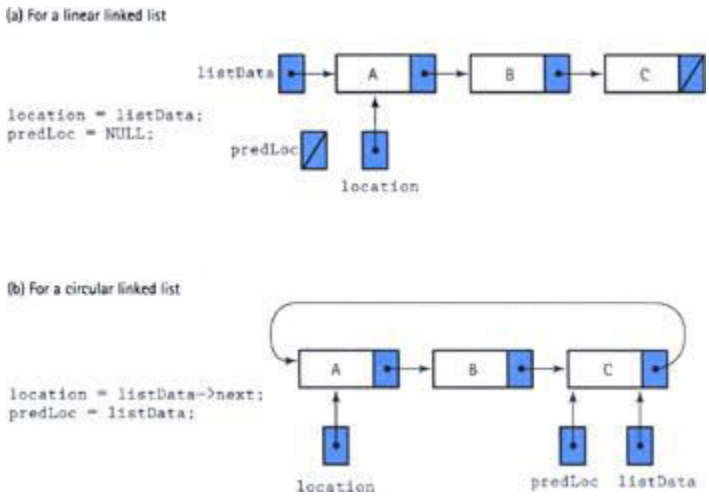


Figure 6.3: Initializing for FindItem

FindItem

```
Set location to Next(listData)
Set predLoc to listData.
Set found to false
Set moreToSearch to true
while moreToSearch AND NOT found DO
   if item < Info(location)
        Set moreToSearch to false
   else if item == Info(location)
        Set found to true
   else
        Set predLoc to location
        Set location to Next(location)
```

```
Set moreToSearch to (location != Next(listData))
```

Following the execution of the loop, if a matching key is found, location points to the list node with that key and predLoc points to its predecessor in the list (Figure 6.4a). Note that if item's key is the smallest key in the list, then predLoc points to its predecessor-the last node in the circular list (Figure 6.4b). If item's key is not in the list, thenpredLoc points to its logical predecessor in the list and location points to its logical successor (Figure 6.4c). Notice that predLoc is correct even if item's key is greater than any element in the list (Figure 6.4d). Thus predLoc is set correctly for inserting an element whose key is larger than any currently in the list.

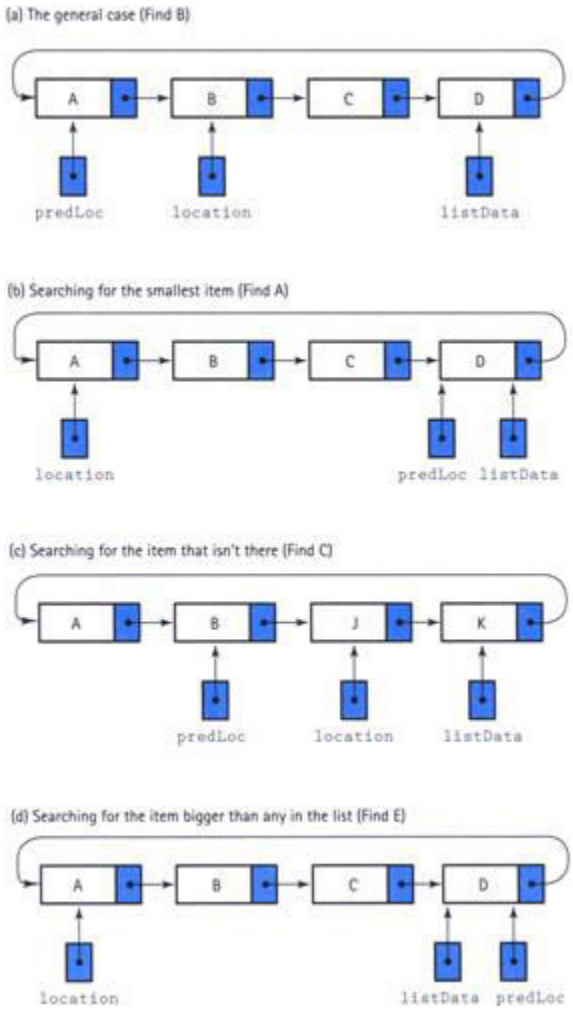


Figure 6.4: The FindItem operation for a circular list

The following C++ code implements our FindItem algorithm as a *function template.* Notice two things in the function heading. First, NodeType was defined in Chapter 5 as a struct template, so each declaration using NodeType must include an actual parameter (the name of a data type) in angle brackets. For example, listData is declared in the code to be of

type NodeType<ItemType>* and not simply NodeType*. Second, observe the syntax for the declarations of location and predLoc. You see an asterisk (*) and an ampersand (&) next to each other. Although this syntax may look strange at first, it is consistent with the usual way we indicate passing-by-reference: Place an & after the data type of the parameter. Here we place an & after the data type NodeType<ItemType>*, which is a pointer to a node.

```
template<claas ItemType>
void FindItem(NodeType<ItemType>* listData, ItemType item,
     NodeType<ItemType>*& location, NodeType<ItemType>*& predLoc,
     bool& found)
// Assumption: ItemType is a type for which the operators "<" and
//  "==" are defined-either an appropriate built-in type or a
//  class that overloads these operations.
// Pre:  List is not empty.
// Post: If there is an element someItem whose key matches item's
//       key, then found = true; otherwise, found = false.
//       If found, location contains the address of someItem and
//       predLoc contains the address of someItem's predecessor:
//       otherwise, location contains the address of item's logical
//       successor and predLoc contains the address of item's
//       logical predecessor.
{
  bool moreToSearch = true;

  location = listData->next:
  predLoc = listData;
  found = false;
  while (moreToSearch && !found)
  {
    if (item < location->info)
      moreToSearch = false;
    else if (item == location->info)
      found = true;
    else
    {
      predLoc = location;
      location = location->next;
      moreToSearch = (location != listData->next);
    }
  }
}
```

Note that FindItem is *not* a member function of the class SortedType. It is an auxiliary or "helper" operation, hidden within the implementation, that is used by SortedTypemember functions.

Inserting Items into a Circular List

The algorithm to insert an element into a circular linked list is similar to that for the linear list insertion.

InsertItem

```
Set new Node to address of newly allocated node
Set Info(newNode) to item
Find the place where the new element belongs
Put the new element into the list
```

The task of allocating space is the same as that carried out for the linear list. We allocate space for the node using the new operator and then store item into newNode-> info. The next task is equally simple; we just call FindItem:

```
FindItem(listData, item, location, predLoc, found);
```

Of course, we do not find the element because it isn't there; it is the predLoc pointer that interests us. The new node is linked into the list immediately after Node(predLoc). To put the new element into the list, we store predLoc->next into newNode->next and newNode into predLoc->next.

Figure 6.5(a) illustrates the general case. What are the special cases? First, we have the case of inserting the first element into an empty list. In this case, we want to makelistData point to the new node, and to make the new node point to itself (Figure 6.5b). In the insertion algorithm for the linear linked list, we also had a special case when the new element key was smaller than any other key in the list. Because the new node became the first node in the list, we had to change the external pointer to point to the new node. The external pointer to a circular list, however, doesn't point to the first node in the list-it points to the last node. Therefore, inserting the smallest list element is not a special case for a circular linked list (Figure 6.5c). However, inserting the largest list element at the end of the list is a special case. In addition to linking the node to its predecessor (previously the last list node) and its successor (the first list node), we must modify the external pointer to point to Node(newNode)-the new last node in the circular list (Figure 6.5d).

The statements to link the new node to the end of the list are the same as those for the general case, plus the assignment of the external pointer, listData. Rather than checking for this special case before the search, we can treat it together with the general case: Search for the insertion place and link in the new node. Then, if we detect that we have added the new node to the end of the list, we reassign listData to point to the new node. To detect this condition, we compare item to listData->info.

The resulting implementation of InsertItem is shown here.

```
template<class ItemType>
void SortedType<ItemType>::InsertItem(ItemType item)
{
  NodeType<ItemType>* newNode;
  NodeType<ItemType>* predLoc;
  NodeType<ItemType>* location;
  bool found;

  newNode = new NodeType<ItemType>;
```

```
  newNode->info = item;
  if (listData != NULL)
  {
    FindItem(listData, item, location, predLoc, found);
    newNode->next = predLoc->next;
    predLoc->next = newNode:

    // If this is last node in list, reassign listData.
    if (listData->info < item)
      listData = newNode;
  }
  else       // Inserting into an empty list.
  {
    listData = newNode;
    newNode->next = newNode;
  }
  length++;
}
```

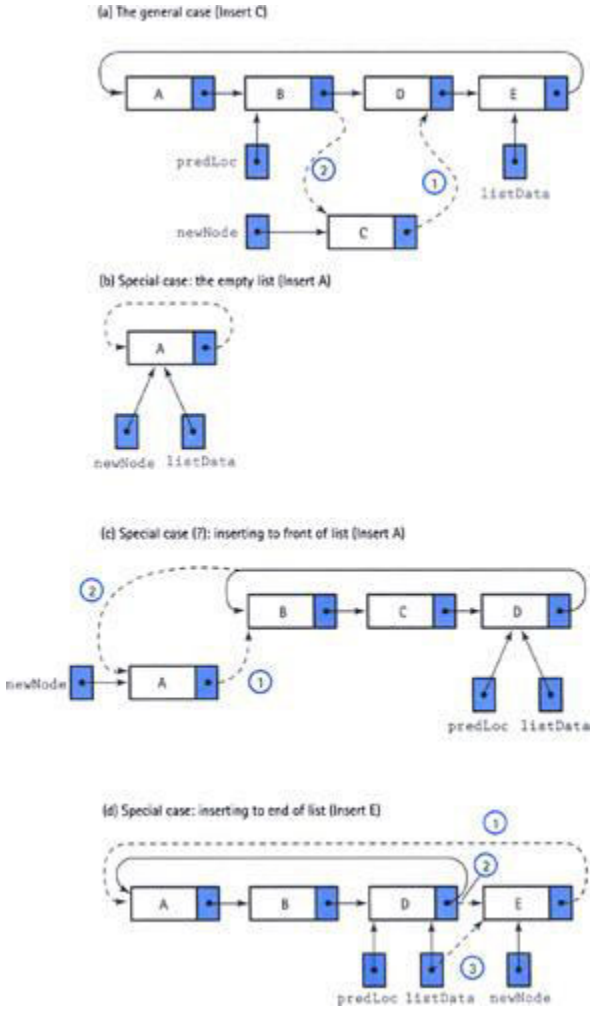


Figure 6.5: Inserting into a circular linked list

Deleting Items from a Circular List

To delete an element from the circular linked list, we use the same general algorithm we developed for the linear list:

DeleteItem

```
Find the element in the list
Remove the element from the list
Deallocate the node
```

For the first task, we use FindItem. After the return from FindItem, location points to the node we wish to delete, and predLoc points to its predecessor in the list. To remove Node(location) from the list, we simply reset predLoc->next to jump over the node we are deleting. That works for the general case, at least (see Figure 6.6a).

What kind of special cases do we have to consider? In the linear list version, we had to check for deleting the first (or first-and-only) element. From our experience with the insertion operation, we might surmise that deleting the smallest element (the first node) of the circular list is *not* a special case; Figure 6.6(b) shows that guess to be correct. However, deleting the only node in a circular list *is* a special case, as we see in Figure 6.6(c). The external pointer to the list must be set to NULL to indicate that the list is now empty. We can detect this situation by checking whether predLoc equals location after the execution of FindItem; if so, the node we are deleting is the only one in the list.

Doubly Linked List example:

Here, we will discuss *deleting an element with a specific value* from a doubly-linked list,

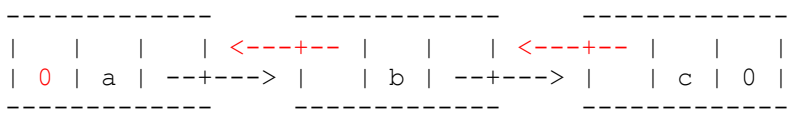
and compare it to deleting from a singly-linked list.

Here are some constraints for our example:

1. The list holds `char`'s.
2. Deletion can occur *anywhere* in the list.
3. The delete operation will not return the element that we delete, it only need remove it from the list.

Now, recall the basic structure of a doubly-linked list:

```
head
 |
 v
```



In other words, there is a node for each element, where nodes consist of a part in which the element is stored, a link to the next node, and a link to the previous node. The last node links to nothing i.e., there are no nodes after it. Also, there are no nodes before the first node. There is a link to the beginning of the list called ***head***.

Difference between an array and Linked list

LINKED LIST:-

1) Can quickly insert and delete items in linked list.
2) You simply rerrange those pointers that are affected by the change in linked list.
3) Linked list are very difficulty to sort.
4) Can not immediately locate the desires element.need to traverse the whole list to reach that element.

5) Linked list are not constrained to be stored in adjacent locations.
6) Basically 3 types are there :- singly,doubly,and circular.
7) Linked list is a list whose order is given by links from one item to the next.

ARRAY:-

1) Can not do so quickly as in linked list.
2) Inserting and deleting items in an array requires you to either make room for new item or fill the hole left by deleting an item.
3) They are not difficulty to sort.
4) Easy to locate the desired element.
5) The element of an array occupy contiguous memory locations.
6) Basically two types are there :- one dimention, two dimention.
7) An array is a group of related data items that share a common name.

Stack :

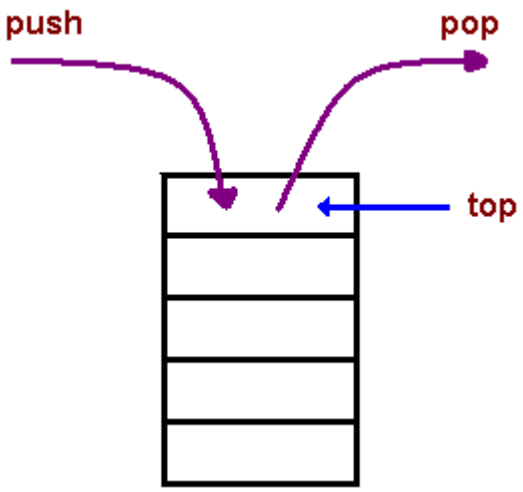
An array is a *random access* data structure, where each element can be accessed directly and in constant time. A typical illustration of random access is a book - each page of the book can be open independently of others. Random access is critical to many algorithms, for example binary search.

A linked list is a *sequential access* data structure, where each element can be accesed only in particular order. A typical illustration of sequential access is a roll of paper or tape - all prior material must be unrolled in order to get to data you want.

In this note we consider a subcase of sequential data structures, so-called *limited access* data sturctures.

Stacks

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: **push** the item into the stack, and **pop** the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. **push** adds an item to the top of the stack, **pop** removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.



A stack is a **recursive** data structure. Here is a structural definition of a Stack:

> a stack is either empty or
> it consistes of a top and the rest which is a stack;

Applications

- The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
- Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.

> **Backtracking**. This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit?

Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.

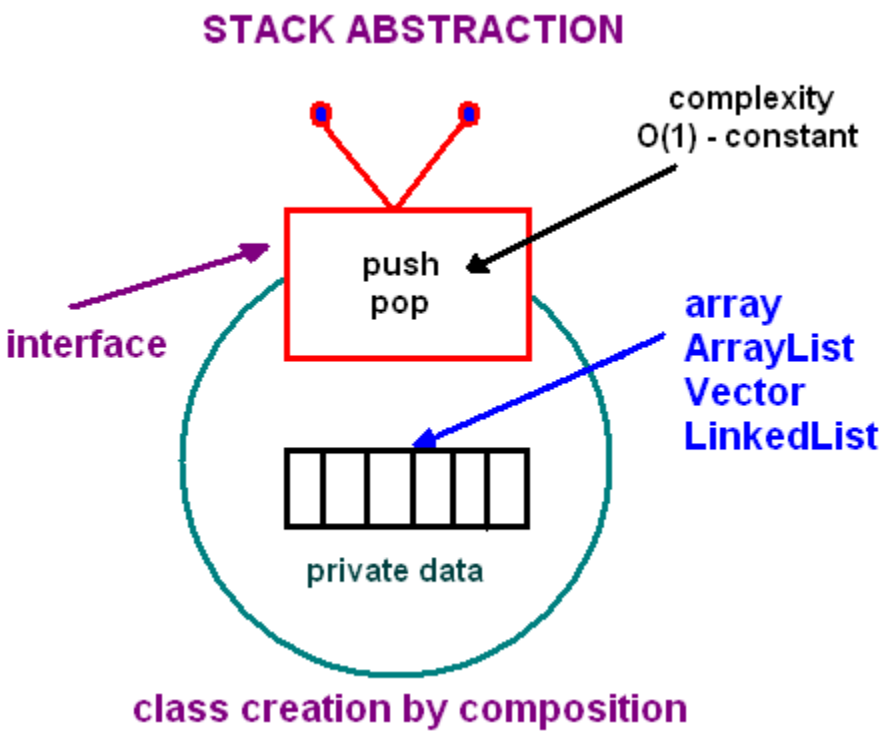
- Language processing:
  - space for parameters and local variables is created internally using a stack.
  - compiler's syntax check for matching braces is implemented by using stack.
  - support for recursion

Implementation

In the standard library of classes, the data type stack is an *adapter* class, meaning that a stack is built on top of other data structures. The underlying structure for a stack could be an array, a vector, an ArrayList, a linked list, or any other collection. Regardless of the type of the underlying data structure, a Stack must implement the same functionality. This is achieved by providing a unique interface:

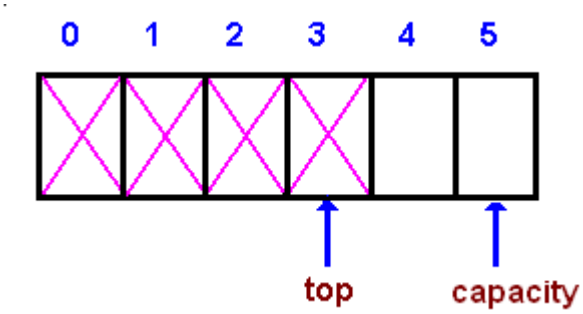
```
public interface StackInterface<AnyType>
{
   public void push(AnyType e);

   public AnyType pop();

   public AnyType peek();

   public boolean isEmpty();
}
```

The following picture demonstrates the idea of implementation *by composition*.



Another implementation requirement (in addition to the above interface) is that all stack operations must run in **constant time O(1)**. Constant time means that there is some constant k such that an operation takes k nanoseconds of computational time regardless of the stack size.

Array-based implementation



In an array-based implementation we maintain the following fields: an array A of a default size (≥ 1), the variable *top* that refers to the top element in the stack and the *capacity* that refers to the array size. The variable *top* changes from -1 to `capacity - 1`. We say that a stack is empty when `top = -1`, and the stack is full when `top = capacity-1`.

In a fixed-size stack abstraction, the capacity stays unchanged, therefore when *top* reaches *capacity*, the stack object throws an exception. See ArrayStack.java for a complete implementation of the stack class.

In a dynamic stack abstraction when *top* reaches *capacity*, we double up the

stack size.

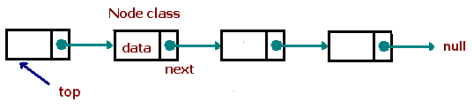
Linked List-based implementation

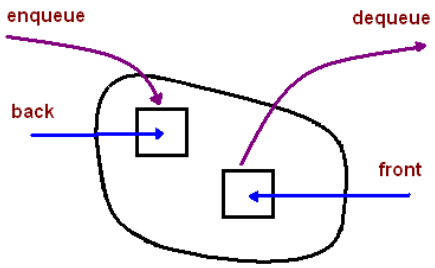
Linked List-based implementation provides the best (from the efficiency point of view) dynamic stack implementation.

See ListStack.java for a complete implementation of the stack class.

Queues

A queue is a container of objects (a linear collection) that are inserted and removed according to the first-in first-out (FIFO) principle. An excellent example of a queue is a line of students in the food court of the UC. New additions to a line made to the back of the queue, while removal (or serving) happens in the front. In the queue only two operations are allowed**enqueue** and **dequeue**. Enqueue means to insert an item into the back of the queue, dequeue means removing the front item. The picture demonstrates the FIFO access.

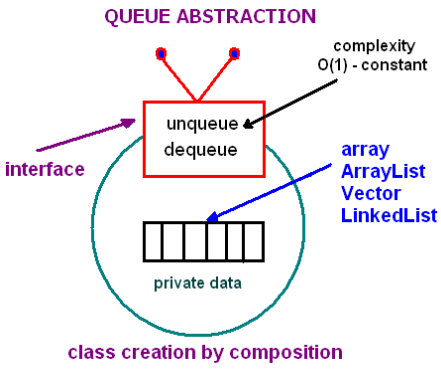




The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Implementation

In the standard library of classes, the data type queue is an *adapter* class, meaning that a queue is built on top of other data structures. The underlying structure for a queue could be an array, a Vector, an ArrayList, a LinkedList, or any other collection. Regardless of the type of the underlying data structure, a queue must implement the same functionality. This is achieved by providing a unique interface.



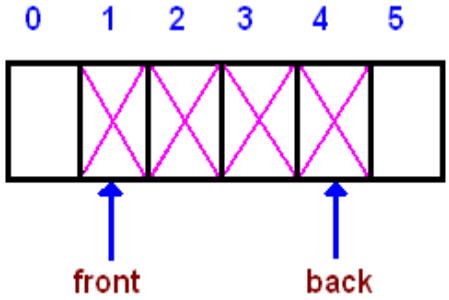

```
interface QueueInterface<AnyType>
{
```

```
    public boolean isEmpty();

    public AnyType getFront();

    public AnyType dequeue();

    public void enqueue(AnyType e);

    public void clear();
}
```

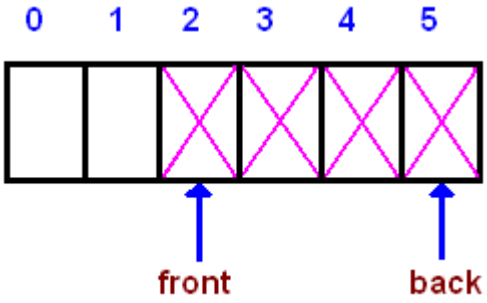
Each of the above basic operations must run at constant time O(1). The following picture demonstrates the idea of implementation by composition.

Circular Queue

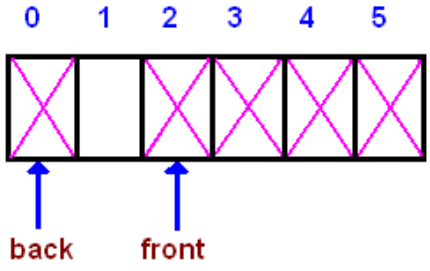
Given an array A of a default size (≥ 1) with two references *back* and *front*, originally set to -1 and 0 respectively. Each time we insert (enqueue) a new item, we increase the back index; when we remove (dequeue) an item - we increase the front index. Here is a picture that illustrates the model after a few steps:



As you see from the picture, the queue logically moves in the array from left to right. After several moves *back* reaches the end, leaving no space for adding new elements



However, there is a free space before the front index. We shall use that space for enqueueing new items, i.e. the next entry will be stored at index 0, then 1, until *front*. Such a model is called a **wrap around queue** or a **circular queue**



Finally, when *back* reaches *front*, the queue is full. There are

two choices to handle a full queue:a) throw an exception; b) double the array size.

The circular queue implementation is done by using the modulo operator (denoted %), which is computed by taking the remainder of division (for example, 8%5 is 3). By using the modulo operator, we can view the queue as a circular array, where the "wrapped around" can be simulated as "back % array_size". In addition to the back and front indexes, we maintain another index: *cur* - for counting the number of elements in a queue. Having this index simplifies a logic of implementation.

Applications

The simplest two search techniques are known as Depth-First Search(DFS) and Breadth-First Search (BFS). These two searches are described by looking at how the search tree (representing all the possible paths from the start) will be traversed.

Deapth-First Search with a Stack

In depth-first search we go down a path until we get to a dead end; then we *backtrack* or back up (by popping a stack) to get an alternative path.

- Create a stack
- Create a new choice point
- Push the choice point onto the stack
- while (not found and stack is not empty)
    - Pop the stack
    - Find all possible choices after the last one tried
    - Push these choices onto the stack
- Return

Breadth-First Search with a Queue

In breadth-first search we explore all the nearest possibilities by finding all possible successors and enqueue them to a queue.

- Create a queue
- Create a new choice point
- Enqueue the choice point onto the queue
- while (not found and queue is not empty)
    - Dequeue the queue
    - Find all possible choices after the last one tried
    - Enqueue these choices onto the queue
- Return

We will see more on search techniques later in the course.

Arithmetic Expression Evaluation

An important application of stacks is in parsing. For example, a compiler must parse arithmetic expressions written using infix notation:

```
1 + ((2 + 3) * 4 + 5)*6
```

We break the problem of parsing infix expressions into two stages. First, we convert from infix to a different representation called postfix. Then we parse the postfix expression, which is a somewhat easier problem than directly parsing infix.

Converting from Infix to Postfix. Typically, we deal with expressions in infix notation

```
2 + 5
```

where the operators (e.g. +, *) are written between the operands (e.q, 2 and 5). Writing the operators after the operands gives a postfix expression 2 and 5 are called operands, and the '+' is operator. The above arithmetic expression is called infix, since the operator is in between operands. The expression

```
2 5 +
```

Writing the operators before the operands gives a prefix expression

```
+2 5
```

Suppose you want to compute the cost of your shopping trip. To do so, you add a list of numbers and multiply them by the local sales tax (7.25%):

```
70 + 150 * 1.0725
```

Depending on the calculator, the answer would be either 235.95 or 230.875. To avoid this confusion we shall use a postfix notation

```
70  150 + 1.0725 *
```

Postfix has the nice property that parentheses are unnecessary.

Now, we describe how to convert from infix to postfix.

1. Read in the tokens one at a time
2. If a token is an integer, write it into the output
3. If a token is an operator, push it to the stack, if the stack is empty. If the stack is not empty, you pop entries with higher or equal priority and only then you push that token to the stack.
4. If a token is a left parentheses '(', push it to the stack
5. If a token is a right parentheses ')', you pop entries until you meet '('.
6. When you finish reading the string, you pop up all tokens which are left there.
7. Arithmetic precedence is in increasing order: '+', '-', '*', '/';

Example. Suppose we have an infix expression:`2+(4+3*2+1)/3`. We read the string by characters.

```
'2' - send to the output.
'+' - push on the stack.
'(' - push on the stack.
'4' - send to the output.
'+' - push on the stack.
'3' - send to the output.
'*' - push on the stack.
'2' - send to the output.
```

Evaluating a Postfix Expression. We describe how to parse and evaluate a postfix expression.

1. We read the tokens in one at a time.
2. If it is an integer, push it on the stack
3. If it is a binary operator, pop the top two elements from the stack, apply the operator, and push the result back on the stack.

Consider the following postfix expression

```
5 9 3 + 4 2 * * 7 + *
```

Here is a chain of operations

```
Stack Operations                Output
---------------------------------------
push(5);                          5
push(9);                          5 9
push(3);                          5 9 3
push(pop() + pop())               5 12
push(4);                          5 12 4
push(2);                          5 12 4 2
push(pop() * pop())               5 12 8
push(pop() * pop())               5 96
push(7)                           5 96 7
push(pop() + pop())               5 103
push(pop() * pop())               515
```

Trees

Trees[

A **tree** is a non-empty set one element of which is designated the root of the tree while the remaining elements are partitioned into non-empty sets each of which is a subtree of the root.

Tree nodes have many useful properties. The **depth** of a node is the length of the path (or the number of edges) from the root to that node. The **height** of a node is the longest path from that node to its leaves. The height of a tree is the height of the root. A **leaf node** has no children -- its only path is up to its parent.

See the axiomatic development of trees and its consequences for more information.

Types of trees:

Binary: Each node has zero, one, or two children. This assertion makes many tree operations simple and efficient.

Binary Search: A binary tree where any left child node has a value less than its parent node and any right child node has a value greater than or equal to that of its parent node.

Traversal

Many problems require we visit the nodes of a tree in a systematic way: tasks such as counting how many nodes exist or finding the maximum element. Three different methods are possible for binary trees: *preorder*, *postorder*, and *in-order*, which all do the same three things: recursively traverse both the left and right subtrees and visit the current node. The difference is when the algorithm visits the current node:

preorder: Current node, left subtree, right subtree (DLR)

postorder: Left subtree, right subtree, current node (LRD)

in-order: Left subtree, current node, right subtree (LDR)

levelorder: Level by level, from left to right, starting from the root node.

* Visit means performing some operation involving the current node of a tree, like incrementing a counter or checking if the value of the current node is greater than any other recorded.

Sample implementations for Tree Traversal

```
preorder(node)
  visit(node)
  if node.left  ≠ null then preorder(node.left)
  if node.right ≠ null then preorder(node.right)
inorder(node)
  if node.left  ≠ null then inorder(node.left)
  visit(node)
  if node.right ≠ null then inorder(node.right)
postorder(node)
  if node.left  ≠ null then postorder(node.left)
  if node.right ≠ null then postorder(node.right)
```

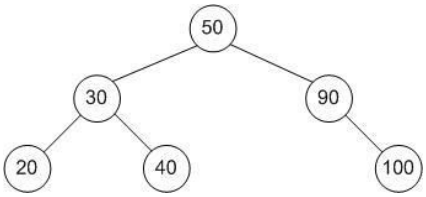
```
  visit(node)
levelorder(root)
  queue<node> q
  q.push(root)
  while not q.empty do
     node = q.pop
     visit(node)
     if node.left  ≠ null then q.push(node.left)
     if node.right ≠ null then q.push(node.right)
```

For an algorithm that is less taxing on the stack, see Threaded Trees.

File:Huma

huma

Examples of Tree Traversals




```
preorder:  50,30, 20, 40, 90, 100
inorder: 20,30,40,50, 90, 100
```

Balancing

When entries that are already *sorted* are stored in a tree, all new records will go the same route, and the tree will look more like a list (such a tree is called a degenerate tree). Therefore the tree needs balancing routines, making sure that under all branches are an equal number of records. This will keep searching in the tree at optimal speed. Specifically, if a tree with *n*nodes is a degenerate tree, the longest path through the tree will be n nodes; if it is a balanced tree, the longest path will be*log n* nodes.

Algorithms/Left_rotation: This shows how balancing is applied to establish a priority heap invariant in a Treap, a data structure which has the queueing performance of a heap, and the key lookup

performance of a tree. A balancing operation can change the tree structure while maintaining another order, which is binary tree sort order. The binary tree order is left to right, with left nodes' keys less than right nodes' keys, whereas the priority order is up and down, with higher nodes' priorities greater than lower nodes' priorities. Alternatively, the priority can be viewed as another ordering key, except that finding a specific key is more involved.

The balancing operation can move nodes up and down a tree without affecting the left right ordering.

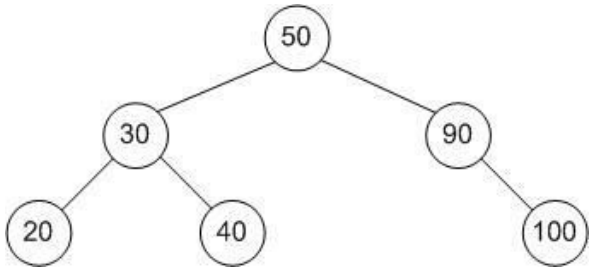
AVL: A balanced binary search tree according to the following specification: the heights of the two child subtrees of any node differ by at most one.

Red-Black Tree: A balanced binary search tree using a balancing algorithm based on colors assigned to a node, and the colors of nearby nodes.

AA Tree: A balanced tree, in fact a more restrictive variation of a red-black tree.

Binary Search Trees

A typical binary search tree looks like this:



Terms

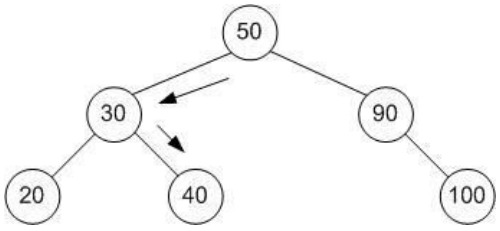
Node Any item that is stored in the tree. **Root** The top item in the tree. (50 in the tree above) **Child** Node(s) under the current node. (20 and 40 are children of 30 in the tree above) **Parent** The node directly above the current node. (90 is the parent of 100 in the tree above) **Leaf** A node which has no children. (20 is a leaf in the tree above)

Searching through a binary search tree

To search for an item in a binary tree:

1. Start at the root node
2. If the item that you are searching for is less than the root node, move to the left child of the root node, if the item that you are searching for is more than the root node, move to the right child of the root node and if it is equal to the root node, then you have found the item that you are looking for.
3. Now check to see if the item that you are searching for is equal to, less than or more than the new node that you are on. Again if the item that you are searching for is less than the current node, move to the left child, and if the item that you are searching for is greater than the current node, move to the right child.
4. Repeat this process until you find the item that you are looking for or until the node doesn't have a child on the correct branch, in which case the tree doesn't contain the item which you are looking for.

Example



For example, to find the node 40...

1. The root node is 50, which is greater than 40, so you go to 50's left child.
2. 50's left child is 30, which is less than 40, so you next go to 30's right child.
3. 30's right child is 40, so you have found the item that you are looking for :)

Adding an item to a binary search tree

1. To add an item, you first must search through the tree to find the position that you should put it in. You do this following the steps above.
2. When you reach a node which doesn't contain a child on the correct branch, add the new node there.

For example, to add the node 25...

1. The root node is 50, which is greater than 25, so you go to 50's left child.
2. 50's left child is 30, which is greater than 25, so you go to 30's left child.
3. 30's left child is 20, which is less than 25, so you go to 20's right child.
4. 20's right child doesn't exist, so you add 25 there :)

Binary Threaded Tree

Threads :

In my previous post I discussed about preorder , postorder and inorder binary tree traversals used stacks and level order traversals used queues as an auxiliary data structure . I highly recommend to know the basic binary tree traversals from the last post .
In this article I will discuss new traversal algorithms which do not need both stacks and queues and such traversal algorithms and called ***Threaded Binary Tree Traversals*** or ***stack/queue less traversals*** .

Threaded Binary Tree Node

Issues with regular Binary Tree Traversals :

- The storage space required for stack and queue is large .
- The majority of pointers in any binary tree are NULL . For example a binary tree with n nodes has n+1 NULL pointers and these were wasted .
- It is difficult to find successor node (preorder ,inorder and postorder successors) for a given node .

Motivation for Threaded Binary Trees :

To solve these problems , one idea is to store some useful information in NULL pointers . If we observe previous traversals carefully , stack/queue is required because we have to record the current position in order to move to right subtree after processing the left subtree . If we store the useful information in NULL pointers then we don't have to store such information in stack / queue. The binary trees which store such information in NULL pointers are called **Threaded Binary Trees** .

The next question is what to store ?

The common convention is pust predecessor/successor information . That means , of we are dealing with preorder traversals then for a given node . NULL left pointer will contain preorder predecessor information and NULL right pointer will contain preorder successor information . These special Pointers are called ***Threads*** .

Classifying Threaded Binary Trees :

The classification is based on whether we are storing useful information in both NULL pointers or only on one of them ,

1. If we store predecessor information in NULL left pointers only then we call such binary trees as ***Left Threaded binary trees*** .
2. If we store successor information in NULL right pointers only then we call such binary trees as ***right threaded binary trees*** .
3. If we store predecessor information in NULL left pointers only then we call such binary trees as ***Fully threaded binary trees*** or simply ***threaded binary trees*** .

Types of Threaded Binary Trees :

Based on above forms we get three representations for threaded binary trees .

1. ***Preorder Threaded Binary Trees*** : NULL left pointer will contain Preorder predecessor information and NULL right pointer will contain preorder successor information .
2. ***Inorder Threaded Binary Trees*** : NULL left pointer will contain Inorder predecessor information and NULL right pointer will contain Inorder successor information .
3. ***Postorder Threaded Binary Trees*** : NULL left pointer will contain Postorder predecessor information and NULL right pointer will contain Postorder successor information .

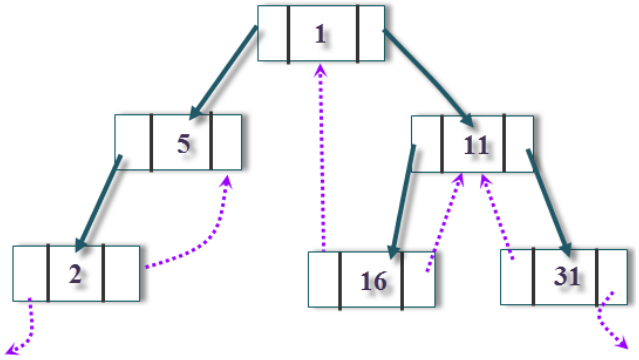
Threaded Binary Tree Structure :

Any program examining the tree must be able to differentiate between a regular left/right pointer and a thread . To do this we use two additional fields onto each node giving us for threaded trees nodes of the following form .

Threaded Binary Tree Node

```
 struct threadedbtnode
{
   struct threadedbtnode *left;
   int Ltag;
   int data;
   int Rtag;
   struct threadedbtnode *right;
};
```

As an example let us try representing a tree in inorder threaded binary tree form . The below tree shows how an inorder threaded binary tree will look like . The dotted arrows indicated the threads . If we observe the left pointer of left most node (2) and right pointer of right most node node (31) are hanging.



Inorder Traversal of a Threaded Binary Tree

To find inorder successor of a given node without using a stack , assume that the node for which we want to find the inorder successor is P.

Strategy : If P has a no right subtree then return the right child if P. If P has right subtree then return the left of the nearest node whose ledt subtree contains P.

```
struct threadedbtnode * inordersuccessor(struct threadedbtnode
*P)

{

    struct threadedbtnode *position;

    if(P->Rtag==0) return P->right;

    else

    {

        position =P->right ;

        while(position->Ltag==1)
```

```
            position=position->left;

        return position;

    }

};
```

Time complexity : O(n)
space complexity : O(1)
Inorder Traversal in Inorder Threaded Binary Tree : We can start with dummy node and call inordersuccessor() to visit each node until we reach **dummy node** .

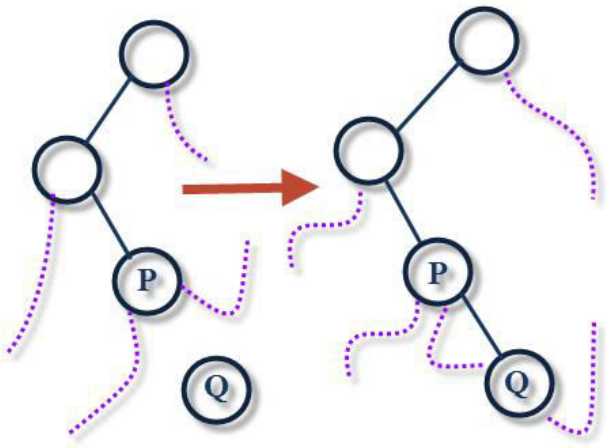
```
void inordertraversal(struct threadedbtnode *root)

{

    struct threadedbtnode *P=inordersuccessor(root);

    while(P!=root)

    {

        P=inordersuccessor(P);

        printf("%d",P->data);

    }

}
```

Time complexity : O(n)
space complexity : O(1)

Inserting a Node into a Threaded Binary Tree

For simplicity let us assume that there are two nodes P and Q and we want to attach Q to right of P. For this we will have 2 cases .

1. **Node P does not has right child :** In this case we just need to attach Q to P and change its left and right pointers .



2. **Node P has right child (say R) :** In this case we need to traverse R's left subtree and find the left most node and then update the left and right pointer of that node . –

Graph

Introduction

In this document, I introduce the concept of a graph and describe some ways of representing graphs in the C programming language.

Definitions

Graphs, vertices and edges

A *graph* is a collection of nodes called *vertices*, and the connections between them, called *edges*.

Undirected and directed graphs

When the edges in a graph have a direction, the graph is called a *directed graph* or *digraph*, and the edges are called *directed edges* or *arcs*. Here, I shall be exclusively concerned with directed graphs, and so when I refer to an edge, I mean a directed edge. This is not a limitation, since an undirected graph can easily be implemented as a directed graph by adding edges between connected vertices in both directions.

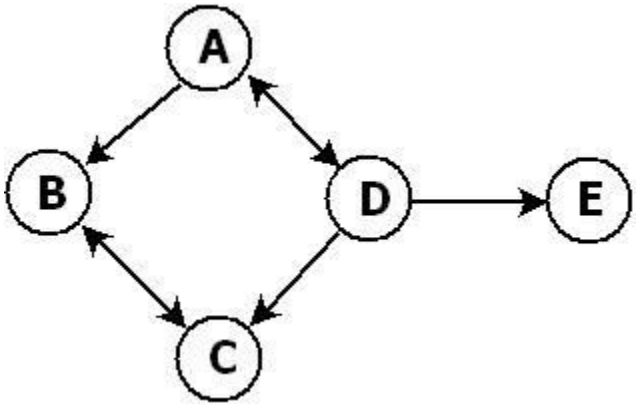
A representation can often be simplified if it is only being used for undirected graphs, and I'll mention in passing how this can be achieved.

Neighbours and adjacency

A vertex that is the end-point of an edge is called a *neighbour* of the vertex that is its starting-point. The first vertex is said to be *adjacent* to the second.

An example

The following diagram shows a graph with 5 vertices and 7 edges. The edges between A and D and B and C are pairs that make a bidirectional connection, represented here by a double-headed arrow.



Mathematical definition

More formally, a graph is an ordered pair, G = <V, A>, where V is the set of vertices, and A, the set of arcs, is itself a set of ordered pairs of vertices.

For example, the following expressions describe the graph shown above in set-theoretic language:

```
V = {A, B, C, D, E}

A = {<A, B>, <A, D>, <B, C>, <C, B>, <D, A>, <D, C>, <D,
E>}
```

Functions

A graph implementation needs a basic set of functions to assemble and modify graphs, and to enumerate vertices, edges and neighbours.

The following functions are provided by each representation. These are the declarations for the intuitive representation, MBgraph1:

```
MBgraph1 * MBgraph1_create(void);
```

Create an empty graph

```
void MBgraph1_delete(MBgraph1 *graph);
```

Delete a graph

```
MBvertex * MBgraph1_add(MBgraph1 *graph, const char *name, void *data);
```

Add a vertex to the graph with a name, and optionally some data

```
MBvertex * MBgraph1_get_vertex(const MBgraph1 *graph, const char *name);
```

Retrieve a vertex by name

```
void * MBgraph1_remove(MBgraph1 *graph, MBvertex *vertex);
```

Remove a vertex

```
void MBgraph1_add_edge(MBgraph1 *graph, MBvertex *vertex1, MBvertex
*vertex2);
```

Create a directed edge between vertex1 and vertex2

```
void MBgraph1_remove_edge(MBgraph1 *graph, MBvertex *vertex1, MBvertex
*vertex2);
```

Remove the directed edge from vertex1 to vertex2

```
unsigned int MBgraph1_get_adjacent(const MBgraph1 *graph, const MBvertex
*vertex1, const MBvertex *vertex2);
```

Determine if there is an edge from vertex1 to vertex2

```
MBiterator * MBgraph1_get_neighbours(const MBgraph1 *graph, const MBvertex
*vertex);
```

Get the neighbours of a vertex

```
MBiterator * MBgraph1_get_edges(const MBgraph1 *graph);
```

Get all of the edges in the graph

```
MBiterator * MBgraph1_get_vertices(const MBgraph1 *graph);
```

Get all of the vertices in the graph

```
unsigned int MBgraph1_get_neighbour_count(const MBgraph1 * graph, const
MBvertex * vertex);
```

Get the count of neighbours of a vertex

```
unsigned int MBgraph1_get_edge_count(const MBgraph1 * graph);
```

Get the count of edges in the graph

```
unsigned int MBgraph1_get_vertex_count(const MBgraph1 * graph);
```

Get the count of vertices in the graph

Representation of vertices and edges

Vertices

All of the graph representations use the following definition of a vertex:

```
typedef struct {
    char * name;
    void * data;
    void * body;
    MBdeletefn deletefn;
} MBvertex;
```

Note the *body* field, which is not of interest to clients, but is used by some representations (Adjacency List and Incidence List) to add per-vertex strucure.

The following functions are provided for working with vertices:

```
const char * MBvertex_get_name(const MBvertex *vertex);
```

Get the vertex's name

```
void * MBvertex_get_data(const MBvertex *vertex);
```

Get the data associated with a vertex

Edges

How edges are implemented internally varies with the representation. In fact, in three representations, Adjacency List, Adjacency Matrix and Incidence Matrix, edges do not exist internally as objects at all. From the viewpoint of clients however, edges, as enumerated by the iterator returned by the function to retrieve edges, are this structure:

```
typedef struct {
    MBvertex *from;
    MBvertex *to;
} MBedge;
```

The following functions are provided for working with edges:

```
const MBvertex * MBedge_get_from(const MBedge * edge);
```

Get the vertex that is the starting-point of an edge

```
const MBvertex * MBedge_get_to(const MBedge * edge);
```

Get the vertex that is the end-point of an edge

Example program

The following program constructs the graph shown in the introduction using the intuitive representation, MBgraph1, and then enumerates the vertices, neighbours and edges:

```
#include <stdio.h>

#include <graph1.h>

int main(void)
{
    MBgraph1 *graph;
    MBvertex *vertex;
    MBvertex *A, *B, *C, *D, *E;
    MBiterator *vertices, *edges;
    MBedge *edge;

    /* Create a graph */
    graph = MBgraph1_create();
```

```
    /* Add vertices */
    A = MBgraph1_add(graph, "A", NULL);
    B = MBgraph1_add(graph, "B", NULL);
    C = MBgraph1_add(graph, "C", NULL);
    D = MBgraph1_add(graph, "D", NULL);
    E = MBgraph1_add(graph, "E", NULL);

    /* Add edges */
    MBgraph1_add_edge(graph, A, B);
    MBgraph1_add_edge(graph, A, D);
    MBgraph1_add_edge(graph, B, C);
    MBgraph1_add_edge(graph, C, B);
    MBgraph1_add_edge(graph, D, A);
    MBgraph1_add_edge(graph, D, C);
    MBgraph1_add_edge(graph, D, E);

    /* Display */
    printf("Vertices (%d) and their neighbours:\n\n",
MBgraph1_get_vertex_count(graph));
    vertices = MBgraph1_get_vertices(graph);
    while ((vertex = MBiterator_get(vertices))) {
        MBiterator *neighbours;
        MBvertex *neighbour;
        unsigned int n = 0;
        printf("%s (%d): ", MBvertex_get_name(vertex),
MBgraph1_get_neighbour_count(graph, vertex));
        neighbours = MBgraph1_get_neighbours(graph, vertex);
        while ((neighbour = MBiterator_get(neighbours))) {
            printf("%s", MBvertex_get_name(neighbour));
            if (n < MBgraph1_get_neighbour_count(graph, vertex) - 1) {
                fputs(", ", stdout);
            }
            n++;
        }
        putchar('\n');
        MBiterator_delete(neighbours);
    }
    putchar('\n');
    MBiterator_delete(vertices);
    printf("Edges (%d):\n\n", MBgraph1_get_edge_count(graph));
    edges = MBgraph1_get_edges(graph);
    while ((edge = MBiterator_get(edges))) {
        printf("<%s, %s>\n", MBvertex_get_name(MBedge_get_from(edge)),
MBvertex_get_name(MBedge_get_to(edge)));
    }
    putchar('\n');
    MBiterator_delete(edges);

    /* Delete */
    MBgraph1_delete(graph);

    return 0;
}
```

Graph representations

There are essentially 5 ways of representing a graph:

- The intuitive representation
- Adjacency List
- Adjacency Matrix
- Incidence Matrix
- Incidence List

The intuitive representation: MBgraph1

What I call the "intuitive" and can also called the "object-oriented" representation is a direct translation of the mathematical definition of a graph into a data type:

```
typedef struct {
    MBset * vertices;
    MBset * edges;
} MBgraph1;
```

- graph1.h
- graph1.c

- Adding a vertex simply requires adding it to the vertex set.
- Adding an edge simply requires adding it to the edge set.
- Removing vertices and edges simply means removing them from the respective sets.
- To find a vertex's neighbours, search the edge set for edges having the vertex as the *from* field.
- To determine if two vertices are adjacent, search the edge set for an edge having the first vertex as its *from* field, and the second vertex as its *to* field.
- Getting all of the edges is easy; just return an iterator over the edge set.
- For undirected graphs, each edge would be stored only once, and getting neighbours and adjacency testing would look at both vertices.
  The edge object would not be *from* and *to* but simply *first* and *second*, i.e., an unordered pair.
- This is one of the representations where edges exist internally as objects (Incidence List is the other).
- This is most like a sparse Adjacency Matrix, with the edge set holding those pairs that are adjacent, and non-adjacent pairs being absent.

Adjacency List: MBgraph2

The graph is made up of a set of vertices. Each vertex contains a set of vertices for its neighbours.

```
typedef struct {
    MBset *vertices;
} MBgraph2;


typedef struct {
    MBset *neighbours;
} vertex_body;
```

- graph2.h
- graph2.c

For the graph shown in the introduction, the sets of neighbours would look like this:

```
A: {B, D}

B: {C}

C: {B}

D: {A, C, E}

E: {}
```

- Adding a vertex just means adding it to the vertex set.
- Adding an edge means adding the end-point of it to the starting vertex's neighbour set.
- It is easy to go from a vertex to its neighbours, because the vertex stores them all.
  Just return an iterator over them.
  This makes the *graph* argument in the function to retrieve neighbours unnecessary in this implementation.
- Testing for adjacency is easy; just search the first vertex's neighbours for the second vertex.
- Getting all edges is more difficult to implement in this representation, because edges don't exist as objects.
  You need to iterate over the neighbours of each vertex in turn, and construct the edge from the vertex and the neighbour.

Adjacency Matrix: MBgraph3

The graph is made up of a set of vertices and a matrix, whose rows and columns are indexed by vertices, and which contains a 1 entry if the vertices are connected.

```
typedef struct {
    MBset    * vertices;
    MBmatrix * edges;
} MBgraph3;
```

- graph3.h
- graph3.c

The adjacency matrix for the graph shown in the introduction would look like this:

```
  ABCDE

A 01010

B 00100

C 01000

D 10101

E 00000
```

- When adding a vertex, add a row and column to the matrix.
- When removing a vertex, remove its row and column.
  As adding and removing rows and columns is expensive, these make the adjacency matrix unsuitable for graphs in which vertices are frequently added and removed.
- Adding and removing edges is easy however, and requires no allocation or deallocation of memory, just setting a matrix element.
- To get neighbours, look along the vertex's row for 1s.
- To determine adjacency, look for a 1 at the intersection of the first vertex's row and the second vertex's column.
- To get the edge set, find all of the 1s in the matrix and construct the edges from the corresponding vertices.

- If the graph is undirected, the matrix will be symmetrical about the main diagonal.
  This means that you can drop half of it, making a triangular matrix.
- The vertex set needs to be ordered so that the index number of vertices can be looked up, or the matrix needs to be a 2-d map keyed by the vertices themselves.
- Memory used for edges is a constant $|V|^2$.
  The best use of this is a graph that is nearly complete, i.e., has a lot of edges.
- The matrix can be sparse; this relates the memory usage more closely to the number of edges.
  It also makes addition and removal of columns easier (no block shifts), but requires renumbering afterwards.
- You can use booleans or bits in the matrix to save memory.

Incidence Matrix: MBgraph4

The graph is made up of a set of vertices and a matrix, as in Adjacency Matrix, but the matrix is vertices × edges, with each column containing two non-zero entries, one for the starting-point vertex and one for the end-point.

```
typedef struct {
    MBset    * vertices;
    MBmatrix * edges;
} MBgraph4;
```

- graph4.h
- graph4.c

The incidence matrix for the graph shown in the introduction looks like this (1 means "from" and 2 means "to"):

```
A 1100200

B 2012000

C 0021020

D 0200111

E 0000002
```

- When you add a vertex, you add a row to the matrix.
- When you add an edge, you add a column to the matrix.
- When you remove a vertex, you need to remove all of the columns containing the vertex from the matrix.
- Getting the edges means iterating over the columns and constructing the edges from the two values.
- To find neighbours, look for 1s in the vertex's row, and in each such column look for the 2 value, which is the neighbour.
- To determine adjacency, find a column containing a 1 in the starting-point vertex's row, and a 2 in the end-point's row.
- For an undirected graph, you have one column per edge, and just the value 1 for "connected", so each column contains two 1s.

Incidence List: MBgraph5

There is a set of vertices as in Adjacency List, but each vertex stores a list of the edges that it is the starting-point of, rather than neighbours.

```
typedef struct {
    MBset * vertices;
} MBgraph5;

typedef struct {
    MBset *edges;
} vertex_body;
```

- graph5.h
- graph5.c

For the graph shown in the introduction, the sets of edges would look like this:

```
A: {<A, B>, <A, D>}

B: {<B, C>}

C: {<C, B>}

D: {<D, A>, <D, C>, <D, E>}

E: {}
```

- Adding a vertex just means adding it to the vertex set.
- Adding an edge means adding it to its starting vertex's edge set.
- Finding if two vertices are adjacent requires searching the first vertex's edge set for an edge containing the second vertex as its *to* field.
- Getting the neighbours requires retrieving them from the pairs in the set of edges for the vertex.
- Getting the edge set requires enumerating each of the vertices' edge sets in turn.
- You can store the edges in the graph object as well as in each vertex.

Net References

http://ebookmaterials.blogspot.in/

http://cse250.wordpress.com
http://olli.informatik.uni-oldenburg.de

http://en.wikibooks.org/

http://younginc.site11.com/

http://cs-people.bu.edu/
http://latest2012.blogspot.in/
http://www.techfinite.net/
http://www.martinbroadhurst.com/

And many other sites use for net references.

Feedback :

Chinmayb07@hotmail.co.uk