A doubly linked list of integers

```
/* intlist.c
   Builds a collection of integers from user input,
   displays it. */
#include <stdio.h>
#include <stdlib.h>
#define bool int
const bool false = 0, true = 1;
/* Node in doubly linked list: */
typedef struct Nodes nodes;
struct Nodes
{
   int data;
   nodes *next,*prev;
};
nodes* node_init(int n);
bool node_link(nodes* node1,nodes* node2);
/* Doubly linked list: */
struct Int_lists
{
   nodes *first,*last;
   int size;
};
typedef struct Int_lists int_lists;
void list_init(int_lists* L);
void list_insert(int_lists *L,int n);
void list_display(int_lists L);
void main(void)
{
   int_lists list;
   int input;
   // Build list from user input, inserting each integer
   // at the head of the list:
   printf("\n\n");
   list_init(&list);
   do {
      printf("Enter a positive integer (< 0 to quit): ");
      scanf("%d",&input);
      if (input > 0)   list_insert(&list,input);
   } while (input > 0);
   list_display(list);
}
// ADT <nodes>:
nodes* node_init(int n)
/* Creates node with data value <n>, returns pointe. */
{
   nodes *new_node = (nodes*)malloc(sizeof(nodes));
   if (new_node != NULL)
   {
      new_node->data = n;
      new_node->prev = new_node->next = NULL;
   }
   else
      printf("Out of memory in <node_init>\n");
   return new_node;
}

bool node_link(nodes* node1,nodes* node2)
/* Links <node1> and <node2> together as predecessor and
 successor, tells whether successful. */
{
   if (node1 != NULL && node2 != NULL)
   {
      node1->next = node2;
      node2->prev = node1;
      return true;
   }
   return false;
}

// ADT <lists>:

void list_init(int_lists* L)
/* Constructor. Creates empty list item */
{
   L->first = L->last = NULL;
   L->size = 0;
}

void list_insert(int_lists *L,int n)
/* Creates a node containing data value <n> and inserts
  it at the end of list <L>. */
{
   nodes* new_node = node_init(n);
   if (new_node == NULL)
      return;
   if (L->first == NULL)
      L->first = L->last = new_node;
   else
   {
      node_link(L->last,new_node);
      L->last = new_node;
   }
   ++L->size;
}

void list_display(int_lists L)
/* Outputs data values of all nodes of list,
 starting with <first>. */
{
   nodes *p_node = L.first;
   int i;

   // Loop through list, displaying data member:
   printf("The list is: \n");
   for (i = 0; i < L.size; ++i)
   {
      printf("%d\n",p_node->data);
      p_node = p_node->next;
   }
}
```

A linked list of strings

```
// strlist.cpp
// Builds a linked list of strings from
// user input, displays it, prompts for
// deletions, displays result.
#include <iostream.h>
#include <string.h>
#include <ctype.h>
#include <conio.h>

class list_nodes
{
   char value[80];
   list_nodes *next;
public:
   list_nodes() { next = NULL; };
   list_nodes(char *s)
      { strcpy(value,s); next = NULL; };
   char *get_value() { return value; };
   list_nodes *get_next() { return next; };
   friend class linked_lists;
};

class linked_lists
{
   list_nodes header;
public:
   linked_lists() { header.next = NULL; };
   void prepend(char *s);
   void delete_node(list_nodes *pred);
   void display();
   list_nodes *first() { return header.next; };
   list_nodes *get_header() { return &header; };
};

void main()
{
   // Build list from user input, and display:
   linked_lists list;
   char input[80];
   cout << endl << endl;
   do
   {
      cout << "Enter a string (* to quit): ";
      cin >> input;
      if (strcmp(input,"*") != 0)
         list.prepend(input);
   }
   while (strcmp(input,"*") != 0);
   list.display();

   // Prompt for deletions and display result:
   list_nodes *p_node = list.first(),
              *p_prev = list.get_header();
   while (p_node != NULL)
   {
      cout << "Delete " << p_node->get_value()
      << "?\n";
      if (toupper(getch()) == 'Y')
         list.delete_node(p_prev);
      else
         p_prev = p_node;
      p_node = p_node->get_next();
   }
   cout << "\nWhat remains is:\n";
   list.display();
}

void linked_lists::prepend(char *s)
// Inserts a node containing data value <s> at
// the beginning of a linked list.
{
   list_nodes *new_node = new list_nodes(s);
   if (header.next == NULL)
      header.next = new_node;
   else
   {
      new_node->next = header.next;
      header.next = new_node;
   }
}
void linked_lists::delete_node(list_nodes *pred)
// Deletes the node following <pred> from list.
{
   // Take no action on null parameter:
   if (! pred || ! pred->next)
      return;

   // Record address of node to be deleted:
   list_nodes *deleted_node = pred->next;

   // Link predecessor of deleted node to
   // deleted node's successor:
   pred->next = deleted_node->next;
}

void linked_lists::display()
// Outputs data values of all nodes of list.
{
   // Counter is stored in pointer:
   list_nodes *p_node = header.next;
   // Loop through list, displaying data
   // member:
   while (p_node != NULL)
   {
      cout << p_node->value << endl;
      p_node = p_node->next;
   }
}
```

Some toolbox code for building a lexical analyzer

The program below supplies the tools necessary to construct a lexical analyzer that will build a linked list of token objects that correspond to the tokens found in a C++ source file.

The program reads a C or C++ source file and writes to the screen a lexical analysis of it. For each lexical token (lexeme) found in the source file, the program displays the line number, location in the line, spelling of the lexeme, and name of its lexical category, such as a particular keyword, a delimiter, an operator, or an identifier.

Program *lexer.cpp* reports what tokens are found and then throws the information awayrather than storing them for later use.

A compiler works by building a collection of token objects, usually a linked list, with a lexical analyzer, and then by applying grammar rules to the linear list in order to form a parse tree or its equivalent. The parse tree shows what elements of a program are components of what other elements.

The program below must be compiled with four files present: *lextoken.cpp, lextoken.h, srcebuf.cpp,* and *srcebuf.h.* If you know how to separately compile source files, you may wish to do so with the library files *lextoken* and *srcebuf*. These define classes used in *lexer.cpp*. These libraries can work like a black box for someone building a data structure to store a list of tokens. You may wish to modify them to your taste. For a C-language version see folder *Lexer C version* in *63271*.

Notice that the essential data necessary to build a linked list of tokens is brought into the I/O loop in *main*. Just write classes or abstract data types for tokens and linked lists of tokens, and build your token list data structure.

```
/* lexer.cpp   Displays lexical analysis of a source file. */
#include "lextoken.cpp"
#include "srcebuf.cpp"

void main()
{
   // Read a source file into a buffer of strings, <prog>:
   const int SPEL_W=30, NAME_W=15, LNNUM_W=6, CHNUM_W= 8, NUM_ROWS=22;
   const char* in_file_name = "lextest.cpp";
   cpp_progs prog(in_file_name);
   if (prog.read())
   {
      cout << "Lexical analysis of " << in_file_name << ":" << endl
         << "Line #  Column  Lex. category    Spelling  " << endl;

      // Get tokens from source-file buffer and display info about each one:
      tokens token;
      prog_locations loc(&prog, 1, 0); // Bookmark in <prog> text
      int token_num = 0;
      while (token.extract(loc))
      {
         cout << setw(LNNUM_W) << token.get_line_num() << setw(CHNUM_W)
            << token.get_char_num() << "    " << setiosflags(ios::left)
            << setw(NAME_W) << lex_categ_name[token.get_lex_categ()]
            << setw(SPEL_W) << token.get_spelling(&prog) << resetiosflags(ios::left)
            << setiosflags(ios::right) << endl;
         token_num++;
         if (token_num % NUM_ROWS == 0)
         {
            cout << "OK to continue?";
            char dummy;
            cin >> dummy;
         }
      }
   }
   else
      cout << in_file_name << " not found" << endl;
```

For source files and test file *xtest.txt*, see *Z: \ Solftware \ DKeil \ 63271 \ Collections*

```
}
```

Sample input (test.txt):

```
void lex_analyze(ifstream& infile);

void main()
{
   cout << "C++ program name? ";
   char file_name[80];
```

Sample output:

```
Lexical analysis of lextest.cpp:
Line #  Column  Lex. category Spelling
Lexical analysis of test.txt:
Line #  Column  Lex. category   Spelling
     1        0  void           void
     1        5  id             lex_analyze
     1       16  lparen         (
     1       17  id             ifstream
     1       25  ampersand      &
     1       27  id             infile
     1       33  rparen         )
     1       34  semi           ;
     2        0  void           void
     2        5  main           main
     2        9  lparen         (
     2       10  rparen         )
     3        0  lbrace         {
     4        1  cout           cout
     4        6  inserter       <<
     4        9  string         "C++ program name? "
     4       29  semi           ;
     5        1  char           char
     5        6  id             file_name
     5       15  lbracket       [
     5       16  num            80
     5       18  rbracket       ]
     5       19  semi           ;
. . .
```

To implement a collection of token objects, it is necessary to declare an array of *tokens* objects or a linked list, e.g.:

```
tokens tokn[100];.
```

Below are constant and type definitions from the header file *lextoken.h*, used by the lexer.

```
enum lex_categories
{
   // Operators, delimiters, constant literals
   tkNone, tkError, tkID, tkNum, tkString,
   tkCharLit, tkHdrFName, tkLBrace, tkRBrace, tkLParen,
   tkRParen, tkLBracket, tkRBracket, tkExtractor,
   tkInserter, tkSemi, tkComma, tkColon, tkPeriod,
   tkAddop, tkAsgnop, tkMulop, tkNot, tkOr, tkAnd, tkRelop,
   tkScope,  tkIncOp, tkPound, tkAmpersand, tkLess, tkGrtr,
   // Keywords:
   tkInclude, tkInt, tkChar, tkFloat, tkIf, tkSwitch, tkWhile,
   tkFor, tkDo, tkStruct, tkClass, tkElse, tkPublic,
   tkPrivate, tkEnum, tkConst, tkReturn, tkStatic,
   tkMain, tkVoid, tkCout, tkCin
};
const lex_categories FIRST_KEYWD =
   tkInclude;
const lex_categories LAST_KEYWD = tkCin;

const char* lex_categ_name[LAST_KEYWD+1] =
{
   "none", "error", "id", "num", "string",
   "char-literal", "hdr-filename", "lbrace",
   "rbrace", "lparen", "rparen", "lbracket",
   "rbracket", "extractor", "inserter",
   "semi", "comma", "colon", "period", "addop", "asgnop",
   "mult-op", "not", "or", "and", "rel-op", "scope-op",
   "incr-op", "pound", "ampersand", "less", "greater",
   // Keywords:
   "include", "int", "char", "float", "if", "switch", "while",
   "for", "do", "struct", "class", "else", "public",
   "private", "enum", "const", "return", "static",
   "main", "void", "cout", "cin"
};

class tokens
{
public:
   tokens();
   bool extract(prog_locations& loc);
   void save(ofstream& outfile,
      cpp_progs prg);
   lex_categories get_less_or_grtr
      (prog_locations& loc, char c);
   lex_categories tokens::keyword_or_ID
      (char* ln, int ch, int len);
   char* get_spelling(cpp_progs* p_prog)
   const;
   lex_categories get_lex_categ() const;
   int get_line_num() const;
   int get_char_num() const;
private:
   int line_num, char_num, length;
   lex_categories category;
};
```

The data members of this class correspond to the columns of output of the program (this page, upper-left).

Symbol tables

The token list specifies that some tokens are identifiers (see column 3 of output above). To build a symbol table, use the spelling of each token that is an identifier. Do not include the same identifier twice in your symbol table. A symbol table is a collection of strings. See diagram below for guidelines on what you will do.

Argument for correctness of Quicksort

To show:

Quicksort sorts an array of any size; i.e., for all i and all j up to the length of array A, such that $i < j$, Quicksort produces an array in which $A[i] \leq A[j]$.

Base step:

Take a partition of size 1. The assertion that Quicksort sorts it is vacuously true since i and j are the same, so there are no i and j such that $i < j$.

Inductive step:

1. We start by assuming that Quicksort always leaves all pairs $A[i]$ and $A[j]$ in an array of a certain size n in order: $A[i] \leq A[j]$.
2. Now consider an array of size $(n + 1)$. The partition step yields three subarrays; the first is unsorted, the second is a subarray of 1 element (the pivot), and the third is unsorted. But the partition step by definition produces a left partition all of whose elements are less than or equal to the pivot, which by definition of the partition step is less than or equal to any of the elements of the right partition.
3. After the partition step, the Quicksort algorithm applies Quicksort to the left and right partitions. Since the pivot is one element, each partition is of size n or less. But by our assumption in (1), we know that an array of size n is sorted successfully by Quicksort, and the base step has shown that an array of size 1 is also sorted successfully. So if partitions are of size n and 1, the array of size $n + 1$ will be sorted by Quicksort.
4. It remains to show that partitions of all other sizes between 1 and n are sorted successfully by the recursive calls to Quicksort. But if any array of size n, $n > 1$, is handled correctly, then we can be inclined to believe that any array of size $(n - 1)$ is handled correctly as well, since it is a simpler task, guaranteeing success for the recursive calls.

Conclusion:

From (3) (4), and (5) in the inductive step, we have shown that if Quicksort sorts any array or subarray of size n, then it also sorts any array or subarray one element larger than n. The base step established that for $n = 1$, Quicksort correctly sorts the array. Thus for all integers n greater than or equal to zero, Quicksort correctly sorts an array of size n.

Note: The above argument by D. Keil does not satisfy the requirements for a rigorous proof. Step 4 of the induction is "hand-waving."

Iterative version of Quicksort

The program below demonstrates Quicksort in an iterative version, using a *while* loop inside a *do…while* loop. The function *iterative_quick* is the one of most interest. This function cannot be built without an auxiliary data structure, consisting of two stacks. This storage replaces the runtime stack, used to handle recursive calls to Quicksort in the standard recursive version.

It might be interesting to compare the running time of the version below with a program that uses a comparable recursive version of Quicksort.

You can understand Quicksort better if you trace its execution. To do this, call the function *display_array* from inside *partition* or *iterative_quick*.

Some trace code is included and commented out.

The program uses different data each time to make up its test array, via a randomization routine in *main*. The current time of day is used to initialize the random-number generator, *rand*.

It might be interesting to see whether the iterative version performs as well with sorted or almost-sorted data as it does with random data.

```
// quiciter.cpp
// Demo of iterative version of Quicksort.
// David Keil, Framingham State College, 2/98
#include <iostream.h>
#include <iomanip.h>
#include <time.h>
#include <stdlib.h>

const int ARR_SZ = 60;

void iterative_quick(int A[],int size);
int partition(int A[],int first,int last);
void swap(int& a,int& b);
void display_array(int A[],int size,
   char message[]);

void main()
{
   // Set up and show random array:
   srand((unsigned int)time(NULL));
  int A[ARR_SZ];
   for (int i=0; i < ARR_SZ; ++i)
      A[i] = rand() % 100;
   display_array(A,ARR_SZ,"Random array:");

   // Sort array and display:
   iterative_quick(A,ARR_SZ);
   display_array(A,ARR_SZ,"Sorted array:");
}
```

```
void iterative_quick(int A[],int size)
// Sorts subarray A[m.n] into ascending
// order using Quicksort algorithm,
// implemented with 'while' loop nested in
// 'do...while' loop. Requires two stacks
// to store pending sub-array endpoints.
{
   // Subscripts in 'A':
   int first=0,last=size;
   // Stacks:
   int left_pending[ARR_SZ],
      right_pending[ARR_SZ],top=0;
   do {
      // Retrieve pending bounds:
      if (top >0)
      {
         top--;
         first = left_pending[top];
         last = right_pending[top];
      }
      // Repeatedly partition sub-arrays:
      while (first < last)
      {
         int pivot = partition(A,first,last);
         left_pending[top] = pivot + 1;
         right_pending[top] = last;
         top++;
         last = pivot - 1;
      }
   } while (top > 0);
}
```

```
int partition(int A[],int first,int last)
// Selects first element of A[first...last] as
// a pivot value, leaves this sub-array in a
// state in which all values less than 'pivot'
// are to its left and all values greater are to
// its right. Returns pivot index.
{
   int pivot = A[first];
   int left = first + 1,
       right = last;
   // Repeatedly find leftmost element in left
   // partition that should be in right and
   // rightmost that should be in left,
   // swap them
   do {
      while (left <= right && A[left] <= pivot)
         ++left;
      while (left <= right && A[right] > pivot)
         --right;
      if (left < right)
         swap(A[left],A[right]);
   } while (left <= right);
   // Move pivot into middle:
   swap(A[first],A[right]);
   return right;
}
```

```
void swap(int& a,int& b)
// Exchanges values of <a>, <b>.
{
   int temp = a;
   a = b;
   b = temp;
}

void display_array(int A[],int size,
   char message[])
// Shows array with 'message'.
{
   const int NUM_COLS = 10;
   cout << endl << message << endl;
   for (int i=0; i < ARR_SZ; ++i)
   {
      cout << setw(5) << A[i];
      if ((i + 1) % NUM_COLS == 0)
         cout << endl;
   }
   cout << endl;
}
```

Counting steps executed by an algorithm

```
// timed_bin_search.cpp
// Searches array of integers using binary-search algorithm, counting steps.
#include <iostream.h>

#define bool int
#define false 0
#define true 1
bool binary_search(int A[],int key,int size);
unsigned count = 0;
void main()
{
   int arr[] = { 2,3,4,6,7,9,10,12,14,17,18,20 },
      key;
   cout << "Enter search key: ";
   cin >> key;
   if (binary_search(arr,key,sizeof(arr)/sizeof(int)))
      cout << "Found" << endl;
   else
      cout << "Not found" << endl;
   cout << "Array is of " << sizeof(arr) / sizeof(int)
      << " elements" << endl;
   cout << "Binary search was done in " << count
      << " comparisons" << endl;
}
bool binary_search(int A[],int key,int size)
// Iterative. Tells whether <key> is present in <A>.
{
   int first = 0, last = size-1, middle;
   bool found = false;
   while (first <= last && ! found)
   {
      middle = (first + last) / 2;
      count++;
      if (A[middle] == key)
         found = true;
      else
         if (key < A[middle])
            last = middle - 1;
         else
            first = middle + 1;
   }
   return found;
}
```

Step counter (annotation on `unsigned count = 0;`)

We define a step in program as one comparison (annotation on `count++;`)

Sample I/O:

```
Enter search key: 4
Found
Array is of 12 elements
Binary search was done in 2 comparisons

Enter search key: 5
Not found
Array is of 12 elements
Binary search was done in 3 comparisons
```

Using the hardware clock

```
// timer.h
// Declares class <timers>, used to time processes.
#include <time.h>
#include <sys\timeb.h>

// Class <elapsed_times>:

class timers
{
public:
    void set_start();
    void set_end();
    double elapsed();
private:
    double start,end;
};
```

```
// timer.cpp
// Defines member functions of class <timers>.
// This class may be used to time processes.

void timers::set_start()
// Assigns current time to <start>.
{
    //    time_t ltime;
    long seconds;
    time(&seconds);
    struct _timeb tstruct;
    _ftime(&tstruct);
    start = seconds + tstruct.millitm/1000.0;
}

void timers::set_end()
// Assigns current time to <end>.
{
    long seconds;
    time(&seconds);
    struct _timeb tstruct;
    _ftime( &tstruct );
    end = seconds + tstruct.millitm/1000.0;
}

double timers::elapsed()
// Returns string representation of time elapsed from
<start> to <end>.
{
    return end-start;
}
```

```
// writeran.cpp
// Generates a file of 100,000 random integers and determines how
// long it takes to write them to a disk file, using
// library functions available with Microsoft Visual C++ 4.0.
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>
#include "timer.h"
#include "timer.cpp"

const long MAX = 100000;

void main()
{
    ofstream outfile("writeran.out");
    cout << "Writing " << MAX
          << " random numbers to file WRITERAN.OUT\n";
    timers stopwatch;
    stopwatch.set_start(); // Declared in <timer.h>
    for (int num_written=0; num_written < MAX; ++num_written)
        outfile << rand() << " ";
    stopwatch.set_end(); // <timer.h>
    cout << "Time to write " << MAX << " integers to disk: "
         << stopwatch.elapsed() << " seconds" << endl;
}
```

A generic stack class using a C++ class template

A C++ *class template* sets a pattern for defining classes that are *parameterized* by type. For example, you may define one general-purpose template for collections of integers, characters, or objects of any type you choose. The C++ *Standard Template Library* (STL) is a library of collection class templates designed for general use.

The program below illustrates the template concept for stacks by defining a class template, *stacks*. In *main,* two classes that are instances of the template are created by the statements *stacks<int> stack1;* and *stacks<char> stack2*. The objects, *stack1* and *stack2*, are instances of stack classes generated by the template. The *stack1* object is a stack of integers and the *stack2* object is a stack of characters. Both stacks have appropriate push and pop member functions. You could define a third or fourth stack to hold a third or fourth type of component of your choice.

```
// genstack.cpp
// Demonstrates use of template class to define generic
// stack class. Pushes a series of integers onto a stack,
// pops them to display in reverse order.
#include <iostream.h>

typedef int bool;

const int MAX_STACK_SZ = 20;

template <class T>
class stacks
{
public:
   stacks() { size = 0; };
   void push(T n)
      { if (!is_full()) element[size++] = n; };
   T pop()
      { return (! is_empty() ? element[--size] : -1); };
   bool is_empty()
      { return (size == 0); };
   bool is_full()
      { return (size >= MAX_STACK_SZ); };
private:
   T element[MAX_STACK_SZ];
   int size;
};

void main()
{
   stacks<int> stack1;
   stack1.push(1);
   stack1.push(2);
   stack1.push(3);
   while (! stack1.is_empty())
      cout << stack1.pop() << endl;
   stacks<char> stack2;
   stack2.push('a');
   stack2.push('b');
   stack2.push('c');
   while (! stack2.is_empty())
      cout << stack2.pop() << endl;
}
```

Output:

```
1
2
3
a
b
c
```

A queue to search an input stream

The program below uses a circular-array implementation of a queue to search an input stream of characters for a match to a pattern. The sample user session below shows the contents of the buffer queue at each user input. First the buffer is filled up, then characters are added to the right end as characters at the left end are removed from the queue. After each character input (at each line below), the contents of the queue are compared, character for character, with the search pattern, “q???e”, where ‘?’ is a wild card character that matches any character in the input stream. Lines of output bracketed with asterisks signal matches to this pattern.

Sample input/output:

```
Enter a stream of characters terminated by *:
T
Th
The
The
The q
The qu
The qui
The quie
The quiet
The quiet
he quiet q
e quiet qu
 quiet que
quiet queu
uiet queue
iet queue
et queue i
t queue is
 queue is
queue is q
queue is q
**** matches q???e ****
ueue is qu
eue is qui
ue is quit
e is quite
 is quite
is quite q
s quite qu
 quite qui
quite quic
quite quic
**** matches q???e ****
uite quick
```

```
/*
   searchq.c
   Maintains most recent 10 characters
   of input stream in a circular queue,
   searching for the pattern "q???e",
   using '?' as a wild card.
   David Keil, Framingham State College, 1/99
*/
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define bool int
const bool false=0,true=1;
// <queues> ADT:
#define MAX_Q_SIZE 10
struct Queues
{
   char element[MAX_Q_SIZE];
   int first,
       size;
};
typedef struct Queues queues;
/* Standard queue interface: */
void q_init(queues* q);
void q_enqueue(queues* q,char c);
char q_dequeue(queues* q);
bool q_is_empty(queues q);
bool q_is_full(queues q);
/* Functions to operate on this
   specific kind of queue of chars: */
void display_queue(queues q);
bool compare_queue(queues q,const char* s);
void main()
{
   const char STOP = '*';
   queues buffer;
   char input;
   const char PATTERN[] = "q???e";
   q_init(&buffer);
   printf("Enter a stream of characters "
      "terminated by %c: ",STOP);
   do {
      input = getch();
      if (input != STOP)
      {
         q_enqueue(&buffer,input);
         display_queue(buffer);
         if (compare_queue(buffer,PATTERN))
         {
            display_queue(buffer);
            printf("** matches %s **\n",PATTERN);
         }
      }
   } while (input != STOP);
}
```

```
void q_init(queues* q)
// Initializes empty queue.
{
   q->size = 0;
   q->first = 0;
}

void q_enqueue(queues* q,char ch)
// Inserts <ch> at rear of queue.
// First dequeues if queue is full.
{
   int subscript;

   if (q_is_full(*q))
      q_dequeue(q);
   subscript =
      (q->first + q->size) % MAX_Q_SIZE;
   q->element[subscript] = ch;
   q->size++;
}

char q_dequeue(queues* q)
// Returns char at front of queue, removes it.
// Returns null on empty queue.
{
   char return_val = '\0';

   if (! q_is_empty(*q))
   {
      return_val = q->element[q->first];
      (q->first)++;
      (q->size)--;
      q->first %= MAX_Q_SIZE;
   }
   return return_val;
}

bool q_is_empty(queues q)
// Tells whether <queue> size is 0.
{
   return (q.size == 0);
};

bool q_is_full(queues q)
// Tells whether <queue> size is maximized.
{
   return (q.size >= MAX_Q_SIZE);
};

/* Functions */

void display_queue(queues q)
// Displays all items in queue.
{
   while(! q_is_empty(q))
      putchar(q_dequeue(&q));
   putchar('\n');
}

bool compare_queue(queues q,const char* s)
// Compares leading characters in queue to <s>,
// using a copy of the actual parameter.
{
   unsigned i=0;
   char ch;
   for (i = 0; i < strlen(s); ++i)
   {
      if (q_is_empty(q))
         return false;
      ch = q_dequeue(&q);
      if (ch != s[i] && s[i] != '?')
         return false;
   }
   return true;
}
```

A program to demonstrate heap operations

The program *charheap.cpp* (C version: *charheap.c*) in the *Heaps* subdirectory of the example files for this course builds a maximum heap from a jumbled array of characters, then extracts the maximum character repeatedly until the heap is empty. At each step of *heapify*, the program displays the current state of the heap.

Running the program a few times can give you a good idea how a heap data structure works.

The code below shows the core of the program. The class *char_heap* uses an array of characters, *A*, and an integer, *size*.

The constructor takes an array of characters as a parameter and calls the *build_heap* operation to enforce the heap property, starting at the second-lowest level of the complete binary tree that is stored in the array, and stepping down to the first element of the array, which is the root of the tree.

The standard heap interface, *is_empty, build_ heap, insert,* and *extract_max*, is implemented in member functions of the class, as is the *heapify* operation used by *build_heap, insert* and *extract_ max*. An extra operation, *display*, shows the complete binary tree graphically as it is manipulated by *heapify*.

```
class char_heap
{
public:
   enum { MAX_HEAP_SIZE = 100 };
   char_heap(const char arr[],unsigned sz);
   void build(bool trace);
   bool is_empty();
   bool insert(char x);
   char extract_max();
   void display(unsigned highlight);
private:
   char A[MAX_HEAP_SIZE];
   unsigned size;
   void heapify(unsigned current);
   unsigned parent(unsigned i);
};
```

The *main* function of the program provides a broad outline of what happens:

```
void main()
{
   cout << "This program builds a heap from an array that begins in random order." << endl
      << "After each 'heapify' operation, the heap is displayed." << endl;
   const char A1[] =
      {'E','P','G','S','X','T','M','A','I','O','R','N','A'};
   char_heap h(A1,sizeof(A1));
   cout << endl
      << "The heap is built. Each node has the heap property for a maximum heap." << endl
      << "Now all the elements of the heap will be extracted one at a time." << endl;
   char A2[14] = "";
   unsigned i = 0;
   do
      A2[i++] = h.extract_max();
   while (! h.is_empty());
   A2[i] = '\0';
   cout << "The following values, in order, were extracted from the heap: " << A2 << endl;
}
```

Exercises

1. Add code to count how many swaps it takes, in terms of heap size, to transform the original array into a heap; how many swaps occur, per item in the heap, in extracting all elements from a heap. Record these for three different array initializations, of different sizes, including the one actually used. Compare the results with predicted O(lg n) time. Disable the calls to *display* if they get in your way.
2. Modify the program to implement a minimum heap of characters.
3. Implement a heap of integers.

An infix-expression evaluator

The program below uses top-down, recursive-descent parsing to scan through an infix expression that contains numeric constants and arithmetic operators. It displays the value of the expression.

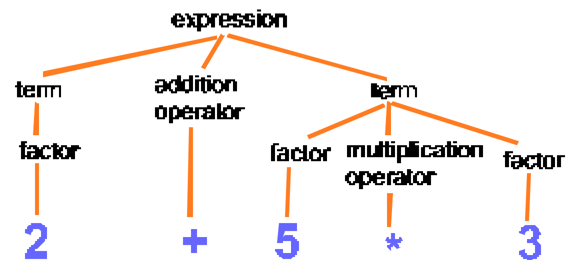
The top level, in *main*, calls the expression-parsing function. That function calls other functions that parse components of an expression, such as terms of an addition or factors of a multiplication.

The functions whose names begin *parse...* each apply a grammar rule of the expression language. Each rule defines a *non-terminal* element of the language.

The rules, which define expressions, terms, and factors, are as follows:

expression → term add-operator term
term → factor || factor mult-operator factor
factor → numeral || (expression)

It is possible to modify the parsing functions to build a *parse tree* of an expression. For example, the parse tree of the expression *2 + 5 * 3* is below.



```
/* infxeval.cpp
  Evaluates infix expression entered
  at keyboard. David Keil, Framingham
   State College, 3/98 */
#include <iostream.h>
#include <ctype.h>
#include <string.h>
typedef int bool;
const bool true = 1,false =0;
const int MAX_LINE_LEN = 80;
class parsers
{
public:
   parsers(char* input);
   int parse_expression();
   int parse_term();
   int parse_factor();
   int parse_numeral();
   char lookahead();
   char next_char();
   char* get_input_line();
   bool at_end();
private:
   char input_line[MAX_LINE_LEN];
   int curr_location;
};
void main()
{
   cout << "Enter an infix expression: ";
   char input[80];
   cin.getline(input,80);
   parsers evaluator(input);
   cout << "The value of your input, "
      << evaluator.get_input_line() << ", is "
      << evaluator.parse_expression() << endl;
}
```

```
// Class <parsers>:

parsers::parsers(char* input)
// Default constructor. Prompts for
// input line.
{
   strcpy(input_line,input);
   curr_location = 0;
}

int parsers::parse_expression()
// expression -> term add_op term
{
   cout << "parsing expression" << endl;
   int return_value = 0;
   return_value = parse_term();
   char op;
   while (((op = lookahead()) == '+' ||
      op == '-') && ! at_end())
   {
      next_char();
      switch (op)
      {
         case '+':
            return_value += parse_term();
            break;
         case '-':
            return_value -= parse_term();
            break;
      }
   }
   return return_value;
}
```

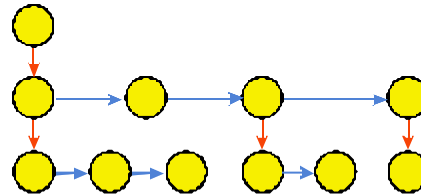
```
int parsers::parse_term()
// term -> factor || factor mult-op factor
{
   cout << "parsing term" << endl;
   int return_value = 0;
   if (lookahead() == '(' ||
      isdigit(lookahead()))
   {
      return_value = parse_factor();
      char op;
      while (((op  = lookahead()) == '*' ||
            op == '/') && ! at_end())
      {
         next_char();
         switch (op)
         {
            case '*':
               return_value *= parse_factor();
               break;
            case '/':
               return_value /= parse_factor();
               break;
         }
      }
   }
   else
      cout << "Expecting ( or digit; found "
         << lookahead() << endl;
   return return_value;
}

int parsers::parse_factor()
// factor -> numeral  || ( expression )
{
   cout << "parsing factor" << endl;
   if (lookahead() == '(')
   {
      next_char();
      int return_value = parse_expression();
      if (lookahead() == ')')
         next_char();
      else
         cout << "Expecting ')'; found "
            << lookahead() << endl;
      return return_value;
   }
   else
      return parse_numeral();
}

int parsers::parse_numeral()
// Advances past numeric constant,
// returns its value.
{
   cout << "parsing numeral" << endl;
   int return_value = 0;
   while (isdigit(lookahead()))
      return_value = return_value * 10 +
         (next_char()-'0');
   return return_value;
}

char parsers::lookahead()
// Returns next character without
// updating counter.
{
   while (isspace(input_line[curr_location]) &&
         curr_location < MAX_LINE_LEN)
      ++curr_location;
   if (curr_location < MAX_LINE_LEN)
      return input_line[curr_location];
   else
      return '\0';
}

char parsers::next_char()
// Returns next character, updates counter.
{
   char c;
   if (curr_location < MAX_LINE_LEN)
   {
      c = lookahead();
      ++curr_location;
   }
   return c;
}

char* parsers::get_input_line()
// Access function.
{
   return input_line;
}

bool parsers::at_end()
// Tells whether input exhausted.
{
   return ((unsigned)curr_location >=
      strlen(input_line)-1);
}
```

Example: General trees

A *general tree* structure may be implemented with nodes that each have two links: one to a child, another to a sibling. Each node is in part of a linked list of siblings. The first child of a node is also part of a linked list of direct descendants of its parent node and any direct ancestors of the parent. Thus each node may in effect have an unbounded number of children, if we consider all the siblings of a node's child to be also children of the node, even if they are not named in *child* links.

A doubly linked general tree will have links both to a node's child and to its parent, to its right sibling and its left sibling. The double linking is not essential but a convenience.

A general tree may be used to implement programming-language parse trees, family trees, taxonomies and organizational structures of all kinds, and outlines. The program below enables the user to create and navigate a free-form outline. The current line of the outline is "highlighted" by brackets. The user may add an item to the outline, as a child of the current node, or may move the highlighting to a different node adjacent to the current one.



```
/*
    outline.c    [subdirectory <General trees>]
    Allows user to build an outline, storing it as a general tree of strings.
    Current node is bracketed. Operations are to [A]dd a node below current one,
    move to [C]hild of current node, move to [P]arent of current node, move to
    [L]eft sibling, or move to [R]ight sibling.

    Node structure of general tree gives each node up to one child and up to one
    right sibling. Each node in effect may have multiple children if its child's
    siblings are considered its children. Each node has four link members because
    it is doubly linked.

    David Keil, Framingham State College, 4/97
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>

typedef int bool;
const bool true=1,false = 0;

// Interface of ADT <nodes>:

typedef struct Nodes nodes;
struct Nodes
{
    char value[80];
    nodes   *child,
            *parent,
            *left_sibling,
            *right_sibling;
};
nodes* node_init(char* val);
```

```
// Interface of ADT <trees>:

struct Trees
{
    nodes *root,
          *current_node;
};
typedef struct Trees trees;

void tree_init(trees* t);
void tree_insert(trees* t,char* new_value,
    nodes* parent);
void tree_display(trees *t,nodes *subtree,
    int level);

// Interface of command ADT:

enum commands {QUIT,ADD,CHILD,PARENT,LEFT,RIGHT};

int get_command();
void exec_command(trees* t,int cmd);
```

```
// <main>:

void main(void)
{
    trees t;
    int command;

    tree_init(&t);
    while ((command = get_command()) != QUIT)
    {
        exec_command(&t,command);
        tree_display(&t,t.root,0);
    }
    printf("Bye.\n");
}

// Implementation of ADT <nodes>:

nodes* node_init(char* val)
// Constructor. Returns a pointer to an
// unlinked node with data value <val>.
{
    nodes* p_node =
        (nodes*)malloc(sizeof(nodes));

    if (p_node == NULL)
    {
        printf("Out of memory in <node_init>\n");
        exit(0);
    }
    strcpy(p_node->value,val);
    p_node->child = p_node->parent =
        p_node->left_sibling =
        p_node->right_sibling = NULL;
    return p_node;
}

// Implementation of ADT <trees>:

void tree_init(trees* t)
// Constructor.
{
    if(t != NULL)
        t->root = t->current_node = NULL;
}
```

```
void tree_insert(trees* t,char* new_value,
    nodes* parent)
/* Creates node with value <new_value>, inserts it into
tree immediately below node <parent>. Creates new root
node if <parent> is null (i.e., tree is empty). */
{
    nodes* p_node = parent;

    if (p_node != NULL)
    {
        // Insert child of <parent>
        // if parent has no child;
        // otherwise add a child to
        // <parent>'s list of children:
        if (p_node->child == NULL)
        {
            p_node->child = node_init(new_value);
            p_node->child->parent = p_node;
            t->current_node = p_node->child;
        }
        else
        {
            p_node = p_node->child;
            while (p_node->right_sibling != NULL)
                p_node = p_node->right_sibling;
            p_node->right_sibling =
                node_init(new_value);
            p_node->right_sibling->left_sibling =
                 p_node;
            t->current_node =
                p_node->right_sibling;
            t->current_node->parent = parent;
        }
    }
    else
        t->root = t->current_node =
             node_init(new_value);
}
```

```
void tree_display(trees *t,nodes *subtree,
    int level)
// Prints subtree through inorder traversal.
// Uses <level> to indent outline.
{
    int i;

    if (subtree != NULL)
    {
        // Indent:
        for (i=0; i < level; ++i)
            printf("   ");
        // Display the subtree and then
        // its siblings: Bracket the node
        // being displayed if it is the
        // tree's current node.
        if (subtree == t->current_node)
            printf("[");
        printf("%s",subtree->value);
        if (subtree == t->current_node)
            printf("]");
        printf("\n");
        if (subtree->child != NULL)
            tree_display(t,
                subtree->child,level+1);
        if (subtree->right_sibling != NULL)
            tree_display(t,
                subtree->right_sibling,level);
    }
}

// Command operations:

int get_command()
// Prompts user for command, returns it.
{
    char input = '\0';

    printf("\nOptions: [Q]uit    "
        "[A]dd a child of current node; \n");
    printf("Move to current node's:   "
        "[C]hild; [P]arent; Sibling: "
        "[L]eft [R]ight ");
    do  {
        scanf("%c",&input);
        switch(toupper(input))
        {
            case 'Q': return QUIT;
            case 'A': return ADD;
            case 'C': return CHILD;
            case 'P': return PARENT;
            case 'L': return LEFT;
            case 'R': return RIGHT;
        }
    } while (!strchr("ACPLRacplr",input));
}
```

```
void exec_command(trees* t,int cmd)
// Executes command <cmd> on tree <t>.
{
    char input[80];

    switch(cmd)
    {
        case ADD:
            printf("Text in new node: ");
            fflush(stdin);
            gets(input);
            tree_insert(t,input,t->current_node);
            break;
        case CHILD:
            if (t->current_node != NULL)
                if (t->current_node->child
                    != NULL)
                    t->current_node =
                      t->current_node->child;
            break;
        case PARENT:
            if (t->current_node != NULL)
                if (t->current_node->parent !=
                     NULL)
                    t->current_node =
                     t->current_node->parent;
            break;
        case LEFT:
            if (t->current_node != NULL)
                if (t->current_node->
                      left_sibling != NULL)
                    t->current_node =
                    t->current_node->left_sibling;
            break;
        case RIGHT:
            if (t->current_node != NULL)
                if (t->current_node->
                      right_sibling != NULL)
                    t->current_node =
                        t->current_node->
                        right_sibling;
            break;
        default:
            printf("Invalid");
    }
}
```

A simple binary search tree of *int*s in C

The program below (see subdirectory *Binary trees*) performs a sort on a file of integers by retrieving the file data into a binary tree and traversing the tree to display them. It uses abstract data types to encapsulate the concepts of nodes and trees, including operations on these types.

The test data for the program, file *intsort.dat*, is as follows:

```
83 61 3 31 37 96 28 15 40 62 53 89 58 18
```

The insertion and display functions, *tree_insert* and *tree_display,* require inorder traversals of the binary tree. Note that these functions are recursive.

```
/*
    treesort.c
    Reads a series of integers from a file,
    stores in binary tree, displays sorted
    list.
*/
#include <stdio.h>
#include <stdlib.h>

typedef int bool;
const bool true=1,false = 0;

// Interface of ADT <nodes>:

typedef struct Nodes nodes;
struct Nodes
{
    int value;
    nodes   *left,
            *right;
};

nodes* node_init(int val);

// Interface of ADT <trees>:

struct Trees
{
    nodes* root;
};
typedef struct Trees trees;

void tree_init(trees* t);
bool tree_retrieve(trees* t,char *filenm);
void tree_insert(trees* t,int new_value,
    nodes* parent);
void tree_display(trees *t,nodes *subtree);
```

```
// <main>:

void main(void)
{
    trees tree;

    tree_init(&tree);
    printf("\nText of file TREESORT.DAT:\n");
    if(tree_retrieve(&tree,"treesort.dat"))
    {
        printf("\nText of same file, %s"
            "sorted numerically: \n");
        tree_display(&tree,tree.root);
    }
}

// Implementation of ADT <nodes>:

nodes* node_init(int val)
// Constructor.
{
    nodes* p_node =
        (nodes*)malloc(sizeof(nodes));

    if (p_node == NULL)
    {
        printf("Out of memory in "
                "<node_init>\n");
        exit(0);
    }
    p_node->value = val;
    p_node->left = p_node->right = NULL;
    return p_node;
}

// Implementation of ADT <trees>:

void tree_init(trees* t)
// Constructor.
{
    if(t != NULL)
        t->root = NULL;
}
```

```
bool tree_retrieve(trees* t,char *filenm)
/* Reads and echoes text file integer by
   integer, stores values in tree using
    <tree_insert>. */
{
    FILE* infile = fopen(filenm,"r");
    int input;

    if (!infile)
    {
        printf("%s not found.\n",filenm);
        return false;
    }
    // Retrieve and insert integers:
    while (fscanf(infile,"%d",&input) != EOF)
    {
        printf("%d ",input);
        tree_insert(t,input,t->root);
    }
    printf("\n");
    fclose(infile);
    return true;
}
```

```
void tree_insert(trees* t,int new_value,
                  nodes* parent)
/* Creates node with value <new_value>,
   inserts it into tree below node <parent>.
   Inserts no new node if <new_value> matches
   an existing node. Creates new root node if
   <parent> is null (i.e., tree is empty). */
{
   if (parent != NULL)
   {
      if (new_value > parent->value)
         if (parent->right)
            tree_insert(t,new_value,
               parent->right);
         else
            parent->right =
               node_init(new_value);
      else
         if (new_value < parent->value)
            if (parent->left)
               tree_insert(t,new_value,
                  parent->left);
            else
               parent->left =
                  node_init(new_value);
   }
   else
      t->root = node_init(new_value);
}

void tree_display(trees *t,nodes *subtree)
// Prints subtree through inorder traversal.
{
   if (subtree != NULL)
   {
      if (subtree->left)
         tree_display(t,subtree->left);
      printf("%d ",subtree->value);
      if (subtree->right)
         tree_display(t,subtree->right);
   }
}
```

A C++ binary search tree of words

```
/*
   treesort.cpp
   Reads all nodes in a text file, stores in
   binary search tree,
   displays sorted list.

   David Keil, Framingham State College, 3/00
*/
#include <iostream.h>
#include <string.h>
#include <fstream.h>
#include <stdlib.h>

class nodes
{
public:
   nodes() { *spelling = '\0'; left = right = NULL; }
   nodes(char *s)
      { strcpy(spelling,s); left = right = NULL; }
   ~nodes() { delete left; delete right; }
   friend class word_trees;
private:
   char spelling[40];
   nodes *left,
        *right;
};

class word_trees
{
public:
   word_trees() { root = NULL; }
   nodes *get_root() { return root; }
   void retrieve(char *filenm);
   void display(nodes *subtree);
   void insert(char *s,nodes *&nd);
private:
   nodes *root;
};

void main()
{
   word_trees dictionary;
   cout << "\nText of file DECL_IND.DAT: \n";
   dictionary.retrieve("decl_ind.dat");
   cout << "\nText of same file, sorted
alphabetically: \n";
   dictionary.display(dictionary.get_root());
}

// Class <word_trees>:

void word_trees::retrieve(char *filenm)
// Reads text file word by word, stores in tree
// using <insert>.
{
   ifstream infile(filenm,ios::nocreate);
   if (! infile)
   {
      cout << filenm << " not found.\n";
      exit(1);  // in <stdlib.h>
   }
   char word[80];
   while (infile >> word)
   {
      cout << word << " ";
      insert(word,root);
   };
   infile.close();
   cout << endl;
}

void word_trees::display(nodes *subtree)
// Prints subtree thru inorder traversal.
{
   if (subtree != NULL)
   {
      display(subtree->left);
      cout << subtree->spelling << " ";
      display(subtree->right);
   }
}

void word_trees::insert(char *s,nodes *&nd)
/* Creates node with string <s>. If not in tree,
   inserts it into tree below node <nd>. */
{
   if (nd != NULL)
   {
      int strdiff = strcmp(s,nd->spelling);
      if (strdiff > 0)
         insert(s,nd->right);
      else
         if (strdiff < 0)
            insert(s,nd->left);
         //else
         // cout << "Already in tree" << endl;
   }
   else
      nd = new nodes(s);
}
```

A binary-search-tree class template

The program below, *intsort.cpp*, uses a library class template to instantiate a class of binary trees of integers.The file *bst.h* that follows defines a class template for binary trees that may be instantiated to build a binary search tree with elements of any type.

```
// intsort.cpp
// Reads all nodes in a text file, stores
// in binary search tree, displays sorted
// list.
// David Keil, Framingham State College,
// 10/98.
#include "bst.h"

void main()
{
   BST<int> tree;
   cout << "\nText of file INTSORT.DAT: \n";
   tree.retrieve("INTSORT.DAT");
   cout << "\nText of same file, sorted: \n";
   tree.display(tree.get_root());
}
```

```
// bst.h
// Defines binary search tree class template.
// David Keil, Framingham State College,
// 10/98.
#include <iostream.h>
#include <string.h>
#include <fstream.h>
#include <stdlib.h>

template <class T>
class BST_nodes
{
public:
   BST_nodes() { left = right = NULL; }
   BST_nodes(T x)
   { data = x; left = right = NULL; }
private:
   T data;
   BST_nodes<T> *left,*right;
   friend class BST<T>;
};
```

```
template <class T>
class BST
{
public:
   BST() { root = NULL; }
   BST_nodes<T> *get_root() { return root; }
   void retrieve(char *filenm);
   void display(BST_nodes<T> *subtree);
   void insert(T x,BST_nodes<T> *&nd);
   BST_nodes<T> *new_node(T x);
private:
   BST_nodes<T> *root;
};

// Class template <BST>:

template <class T>
void BST<T>::retrieve(char *filenm)
// Reads text file word by word, stores in
// tree using <insert>.
{
   ifstream infile(filenm,ios::nocreate);
   if (! infile)
   {
      cout << filenm << " not found.\n";
      exit(1);  // in <stdlib.h>
   }
   T input;
   while(infile >> input)
   {
      cout << input << " ";
      insert(input,root);
   }
   infile.close();
   cout << endl;
}

template <class T>
void BST<T>::display(BST_nodes<T> *subtree)
// Prints subtree thru inorder traversal.
{
   if (subtree != NULL)
   {
      if (subtree->left)
         display(subtree->left);
      cout << subtree->data << " ";
      if (subtree->right)
         display(subtree->right);
   }
}
```

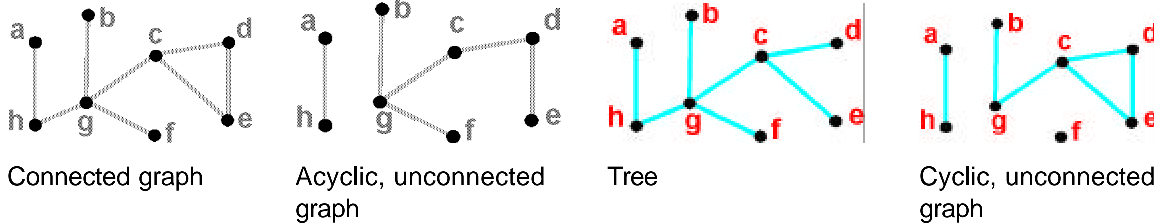
```
template <class T>
void BST<T>::insert(T x,BST_nodes<T> *&p_n)
// Creates node with value <x>,
// inserts it into tree below node <p_n>
// if it is not found in tree.
{
   if (p_n)
   {
      if (x > p_n->data)
         if (p_n->right)
            insert(x,p_n->right);
         else
            p_n->right = new_node(x);
      else
         if (x < p_n->data)
            if (p_n->left)
               insert(x,p_n->left);
            else
               p_n->left = new_node(x);
   }
   else
      p_n = new_node(x);
}
```

```
template <class T>
BST_nodes<T> *BST<T>::new_node(T x)
// Allocates node, assigns <x> to it.
{
   BST_nodes<T> *p_node = new BST_nodes<T>;
   p_node->data = x;
   return p_node;
}
```

Undirected graphs implemented as matrices

The program below, *grafthry.c*, displays matrices that represent undirected graphs and tells whether a path exists between two given vertices in a graph. The problems at the end require extensions to this program.

The header file *grafthry.h* defines a structure type, *matrices,* to encapsulate undirected graphs and declares some instances of the matrix type. The illustrations below show instances of four categories of graph. The matrix for each example is initialized as a matrix in *grafthry.h*.



Connected graph | Acyclic, unconnected graph | Tree | Cyclic, unconnected graph

```
const matrices connected = {
//  A B C D E F G H
  {{0,0,0,0,0,0,0,1}, //A
    {0,0,0,0,0,0,1,0}, //B
    {0,0,0,1,1,0,1,0}, //C
    {0,0,1,0,1,0,0,0}, //D
    {0,0,1,1,0,0,0,0}, //E
    {0,0,0,0,0,0,1,0}, //F
    {0,1,1,0,0,1,0,1}, //G
    {1,0,0,0,0,0,1,0}}, //H
    8,"connected"};

const matrices acyclic_unconnected = {
//  A B C D E F G H
  {{0,0,0,0,0,0,0,1}, //A
    {0,0,0,0,0,0,1,0}, //B
    {0,0,0,1,0,0,1,0}, //C
    {0,0,1,0,1,0,0,0}, //D
    {0,0,0,1,0,0,0,0}, //E
    {0,0,0,0,0,0,1,0}, //F
    {0,1,1,0,0,1,0,0}, //G
    {1,0,0,0,0,0,0,0}},//H
    8,"acyclic unconnected"};

const matrices cyclic_unconnected = {
//  A B C D E F G H
  {{0,0,0,0,0,0,0,1}, //A
    {0,0,0,0,0,0,1,0}, //B
    {0,0,0,1,1,0,1,0}, //C
    {0,0,1,0,1,0,0,0}, //D
    {0,0,1,1,0,0,0,0}, //E
    {0,0,0,0,0,0,0,0}, //F
    {0,1,1,0,0,0,0,0}, //G
    {1,0,0,0,0,0,0,0}},//H
    8,"cyclic unconnected"};

const matrices tree = {
//  A B C D E F G H
  {{0,0,0,0,0,0,0,1}, //A
    {0,0,0,0,0,0,1,0}, //B
    {0,0,0,1,1,0,1,0}, //C
    {0,0,1,0,0,0,0,0}, //D
    {0,0,1,0,0,0,0,0}, //E
    {0,0,0,0,0,0,1,0}, //F
    {0,1,1,0,0,1,0,1}, //G
    {1,0,0,0,0,0,1,0}},//H
    8,"tree"};
```

The structure type *matrices,* and related constants, are defined as follows:

```
#define WIDTH 8
#define DEPTH 8
typedef int BOOLEAN; // This code will use {1,0} rather than
          // {TRUE,FALSE} for reasons of display clarity.
struct Matrices
{
    BOOLEAN pair[WIDTH][DEPTH];        // maximum capacity
    int size;                          // actual width/depth
    char* name;
};
typedef struct Matrices matrices;
```

A matrix is implemented with a *WIDTH* ¥ *DEPTH* array of truth values. In graph-theory related usage, the matrix may be either the adjacency matrix of an undirected graph (the set of ordered pairs of vertices,

each standing for an edge), or else the grid representing the set of paths that exist in the graph.

The main program is as follows. It prompts the user for one of four examples shown above, displays the adjacency matrix of the example, and prompts for a query about the example. Coding the operations called for by the queries is left as an exercise.

```
/*
    grafthry.c
    Allows user to choose an undirected graph
    from samples stored, display its adjacency
    matrix, and display a matrix showing
    all paths in the graph.
*/
#include "grafthry.h"

const matrices* graph_choose();
void graph_show_pairs(matrices graph);
void choose_operation(matrices graph);

void main(void)
{
    int operation;

    matrices* p_m, graph;
    do  {
        p_m = graph_choose();
        if (p_m)
        {
            graph = *p_m;
            graph_show_matrix(graph);
            choose_operation(graph);
        }
    } while (p_m != NULL);
}

const matrices* graph_choose()
// Permits user to select a matrix
// from sample constants, returns
// pointer to it or NULL.
{
    int i,input;

    printf("\n\n\nChoose one graph:\n");
    printf(" 0 Quit\n");
    for (i = 0; i < NUM_EXAMPLES; ++i)
        printf("%2d %s\n",i+1,EXAMPLE[i]->name);
    do
        scanf("%d",&input);
    while (input < 0 || input > NUM_EXAMPLES);
    if (input == 0)
        return NULL;
    else
        return EXAMPLE[input-1];
}
```

```
void graph_show_matrix(matrices graph)
// Displays adjacency matrix <graph>.
{
   int row, column;
   printf("\nThese pairs are in graph"
      "\"%s\"\n",    graph.name);
   for (row = 0; row < graph.size; ++row)
   {
      for (col = 0; col < graph.size; ++col)
         printf("%2d ", graph.pair[row][column]);
      printf("\n");
    }
}

void choose_operation(matrices graph)
// Prompts user for operation to perform on
// <graph>, executes it.
{
    int i, input, num_operations =
        sizeof(OPERATION_NAME) / sizeof(char*);
    printf("\n\nChoose an operation on"
    "   \"%s\":\n", graph.name);
    printf(" 0 Quit\n");
    for (i = 0; i < num_operations; ++i)
        printf("%2d %s\n", i+1, OPERATION_NAME[i]);
    do
        scanf("%d", &input);
    while (input < 0 || input > num_operations);
    switch(input)
    {
        case LIST_EDGES:
            break;
        case FIND_PATH:
            break;
        case FIND_SHORTEST_PATH:
            break;
        case FIND_CYCLE:
            break;
        case FIND_CONNECTED:
            break;
        case FIND_CONNECTIVITY_NUMBER:
            break;
        case FIND_TREE:
            break;
        case FIND_ANCESTORS:
            break;
        case FIND_SHORTEST_WEIGHTED_PATH:
            break;
        case FIND_SPANNING_TREE:
            break;
    }
}
```

The Dijkstra algorithm

The problem of scheduling optimal routes for people, goods, or information to travel is equivalent to the problem of finding the shortest path from one vertex to another in a weighted, directed graph. The algorithm by the computer scientist Edsger Dijkstra is one of those that solves this problem. (See slides.) The program below demonstrates that algorithm using a particular graph, specified by constant *weights*.

```
/* dijkstra.cpp
 Demonstrates action of Dijkstra's algorithm
 for finding shortest path from a single source
 vertex in a weighted directed graph.
 Program could be made more general by reading
 edge weights from a file.  David Keil, 12/97 */
#include <iostream.h>
#include <iomanip.h>
#include <limits.h>
#include <string.h>
#include <stdlib.h>

typedef int boolean;
const boolean false=0,true=1;
const int MAX_VX = 10, // maximum # of vertices
        INF = INT_MAX, NIL = -1;
typedef int vertices;
typedef int weights;
enum VERTICES {S,U,V,X,Y};
const char *name[] = {"S","U","V","X","Y"};

void pause();
int name_to_int(char name);

class int_lists
{
public:
   int_lists() { size = 0; }
   int_lists(int sz,int v[]);
   void set(int sz,int v[]);
   int get_size() { return size; };
   boolean insert(int v);
   void delete_element(int idx);
   void display(char* label);
   void display_edges(char* label);
protected:
   vertices elt[MAX_VX];
   int size;
};

weights A1[MAX_VX * MAX_VX] = {
    // S   U   V   X   Y
      0,   3, INF,  5, INF, // S
     INF,  0,  6,  2, INF,  // U,
     INF,INF,0,  INF,  2,   // V
     INF,  1,  4,   0,  6,  // X
     3,  INF,  7, INF,  0   // Y
     };

class queues : public int_lists
// priority queue, ordered by distance
// from start
{
public:
   queues(int sz,int v[],weights* d);
   int extract_min();
   vertices get_vertex(int i);
private:
   weights* distance;
};

class graphs
{
public:
   graphs() { num_vx = 0; }
   graphs(weights q[],int n,boolean trace);
   int_lists find_min_paths(vertices start);
   void initialize_single_source
      (vertices source);
   boolean relax(vertices u,vertices v);
   void display_wts();
   void display_d(vertices start);
private:
   // Implementation: weighted adjacency matrix
   weights wt[MAX_VX][MAX_VX];
   int num_vx;
   weights d[MAX_VX];// Upper bound on path wt.
   vertices p[MAX_VX];     // predecessor in path
   boolean trace;
};

void main()
{
   /* Data used by this program is <weights>, below,
defining a weighted graph, and the parameter to
<find_min_paths>, called below, which tells the source.
The program displays the set of lengths of minimum paths
from <source> to each of the nodes. */
   graphs G(A1,5,true);
   G.display_wts();
   char input;
   int start;
   do {
      cout << "Starting from where (s,u,v,x,y;"
            << "other to quit)? ";
      cin >> input;
      start = name_to_int(input);
      if (start != NIL)
      {
         int_lists min_paths =
            G.find_min_paths(start);
         G.display_d(start);
      }
   } while (start != NIL);
}
```

```
// Class <graphs>:

graphs::graphs(weights m[],int n,int tr)
// Initializes set of <n> vertices to 2d array
// <wt> from 1D array <m>.
{
   num_vx = n;
   for (int i=0; i < num_vx; ++i)
      for (int j=0; j < num_vx; ++j)
         wt[i][j] = m[num_vx * i + j];
   trace = tr;
}

void graphs::display_d(vertices start)
// Shows set of distances from source.
{
   cout << "Minimal distances from "
      << name[start] << endl;
   for (int j=0; j < num_vx; ++j)
         cout << setw(4) << name[j];
   cout << endl;
   for (int i=0; i < num_vx; ++i)
      if (d[i] == INF)
         cout << setw(4) << "INF";
      else
         cout << setw(4) << d[i];
   cout << endl << endl;
}

int_lists graphs::find_min_paths(vertices start)
/* Dijkstra breadth-first algorithm, returning length of
shortest weighted path from <start> to each other vertex.
*/
{
   vertices intval[] = {S,U,V,X,Y};
   int_lists s;    // set of vx in a min path
   queues q(num_vx,intval,d); // priority queue
   initialize_single_source(start);
   int num_steps = q.get_size();
   for (int step=0; step < num_steps; ++step)
   {
      int origin = q.extract_min();
      for (int dest=0; dest < num_vx; ++dest)
      {
         if (wt[origin][dest] > 0 &&
             wt[origin][dest] < INF)
         {
            boolean did_relax =
               relax(origin,dest);
            if (did_relax)
               s.insert(origin);
         }
      }
   }
   return s;
}

void graphs::initialize_single_source
   (vertices source)
// Initializes vectors used by algorithm
{
   for (int i=0; i < num_vx; ++i)
   {
      d[i] = INF;
      p[i] = NIL;
   }
   d[source] = 0;
}

boolean graphs::relax(vertices orig,
          vertices dest)
/* Lowers upper bound <d> on min path from source to
<dest> if path weights from start to <orig>, and <orig> to
<dest>, are less than current lower bound. */
{
   int origdest = wt[orig][dest],dorig=
         d[orig],ddest=d[dest];
   int test_wt = (origdest == INF ||
         dorig == INF ? INF : dorig + origdest);
   boolean do_relax = d[dest] > test_wt;
   if (do_relax)
   {
      d[dest] = test_wt;
      p[dest] = orig;
   }
   if (trace && do_relax)
   {
      cout << "Relaxing " << name[dest]
         << " from "
         << (ddest == INF ? 9999 : ddest)
         << " to " << d[dest] << endl;
   }
   return do_relax;
}

void graphs::display_wts()
// Shows weights of all edges in graph.
{
   cout << "Weights:" << endl << setw(3) << "";
   for (int i=0; i < num_vx; ++i)
      cout << setw(3) << name[i];
   cout << endl;
   for (i=0; i < num_vx; ++i)
   {
      cout << setw(3) << name[i];
      for (int j=0; j < num_vx; ++j)
      {
         cout << setw(3);
         if (wt[i][j] == INF)
            cout << "";
         else
            cout << wt[i][j];
      }
      cout << endl;
   }
   cout << endl;
}

// Class <int_lists>:

int_lists::int_lists(int sz,int v[])
```

```
// Initializes array <vx> to <v>, size to <sz>.
{
   set(sz,v);
}
void int_lists::set(int sz,int v[])
{
   size = sz;
   for (int i=0; i < size; ++i)
      elt[i] = v[i];
}
boolean int_lists::insert(int v)
// Adds <v> to queue. Tells whether successful.
{
   if (size < MAX_VX-1)
   {
      elt[size] = v;
      size++;
      return true;
   }
   else
      return false;
}
void int_lists::delete_element(int idx)
{
   while (idx < size-1)
   {
      elt[idx] = elt[idx+1];
      idx++;
   }
   size--;
}
void int_lists::display(char* label)
// Shows labelled list of vertices.
{
   cout << label << endl;
   for (int i=0; i < size; ++i)
      cout << setw(4) << elt[i];
   cout << endl;
}

void int_lists::display_edges(char* label)
// Shows labelled list of names of pairs
// of vertices.
{
   cout << label << endl;
   for (int j=0; j < size; j+=2)
      if (elt[j] < 6)
         cout << name[elt[j]] << "-"
            << name[elt[j+1]] << ", ";
      else
         cout << "Out of bounds\n";
   cout << endl;
}
```

```
// Class <queues>:

queues::queues(int sz,int v[],weights* d)
{
   set(sz,v);
   distance = d;
}

vertices queues::get_vertex(int i)
// Access function
{
   if (i >= 0 && i < size)
      return elt[i];
   else
      return NIL;
}

int queues::extract_min()
// Returns element of list that is subscript of
// minimum element of <d>; this is, tells what
// element of list has shortest path so far
// from source referenced by <d>.
{
   if (get_size() < 1)
      return NIL;
   int min_idx=0;
   for (int i=1; i < get_size(); ++i)
   {
      if (distance[get_vertex(i)] <
            distance[get_vertex(min_idx)])
         min_idx = i;
   }
   int return_val = elt[min_idx];
   delete_element(min_idx);
   return return_val;
}

// Utility:
int name_to_int(char name)
{
   switch(toupper(name))
   {
      case 'S': return 0;
      case 'U': return 1;
      case 'V': return 2;
      case 'X': return 3;
      case 'Y': return 4;
      default: return NIL;
   }
}

void pause()
{
   char dummy[10];
   cout << "Press any char and <return>";
   cin >> dummy;
}
```

A hash table minus collision resolution

The program below gives basic tools for implementing open addressing, or the linear-probe strategy for resolving collisions. The program implements a hash table, using an array of pointers to strings that is a member of class *hash_tables*. The hash table will be incomplete because all collisions will be resolved by omitting the new value from the table.

To experiment with the program, change the data file you read to a C++ program, for example, or another text file; change the array size, set in the *SIZE* constant declared as an enumerated constant in class *hash_tables*; see how many collisions occur on each table slot; and change the hash function used from *sum_of_letters* to *mid_square* or some other function of your choice.

To implement collision resolution, you must modify *hash_tables* member functions *insert* and *slot*. The *insert* function must find the next open slot and insert the new element there. In searching for an entry in the table, the *slot* function must not report failure until it has found an empty slot after looking unsuccessfully at a series of slots starting with the slot indicated by the hash function.

```
/*
   hashdemo.cpp
   Reads distinct strings from a text file,
   stores non-colliding ones in a hash table,
   displays table. Since program does not
   handle collisions, the resulting hash table
   is incomplete.
   David Keil, Framingham State College, 2/29/98
*/
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <string.h>

const int NOT_FOUND = -1;
int mid_square(char* s);
int sum_of_letters(char* s);

typedef int bool;
const bool false=0,true=1;

class hash_tables
{
public:
   hash_tables();
   void insert(char* s);
   bool contains(char* s) const;
   void display() const;
   int get_collisions() const;
            private:
               int hash(char* s) const;
               int slot(char* s) const;
   enum { SIZE = 80 };
   char* element[SIZE]; // Slot in hash table
   int num_collisions;  // Temporary member
};
```

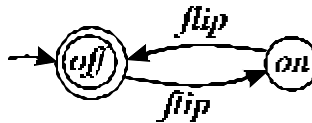
Hash table

```
void main()
{
   const char* FILE_NAME = "decl_ind.txt";
   hash_tables ht;
   ifstream infile(FILE_NAME,ios::nocreate);
   if(! infile)
      cout << FILE_NAME << "not found." << endl;
   else
   {
      int word_count = 0,num_distinct = 0;
      while (! infile.eof())
      {
         char word[256];
         infile >> word;
         cout << word << " ";
         char* comma = strchr(word,',');
         // Remove commas after nodes:
         if (comma)
            *comma = '\0';
         if (! ht.contains(word))
         {
            ++num_distinct;
            ht.insert(word);
         }
         ++word_count;
      }
      cout << endl;
      ht.display();
      cout << "Above is a hash table built "
         << "from the " << word_count
         << " words (" << num_distinct
         << " distinct) in " << FILE_NAME
         << "." << endl << "In the cases of "
         << ht.get_collisions()
         << " collisions, no entry was inserted."
         << endl;
   }
}

// Class <hash_tables>:
```

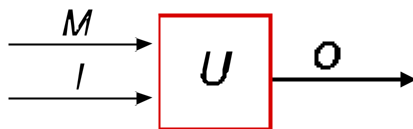
```
hash_tables::hash_tables()
// Default constructor.
{
   for (int i=0; i < SIZE; ++i)
      element[i] = NULL;
   num_collisions = 0;
}

void hash_tables::display() const
// Displays all values in the table with their
// subscripts.
{
   cout << "Ready to see hash table?";
   char ans;
   cin >> ans;
   const int NUM_COLS = 4, COL_WIDTH = 12;
   for (int i=0; i < SIZE; ++i)
   {
      cout << setw(4) << setiosflags(ios::right)
            << i << " " << setw(COL_WIDTH)
            << setiosflags(ios::left);
      if (element[i] != NULL)
         cout << element[i];
      else
         cout << "[empty]";
      cout << resetiosflags(ios::left);
      if ((i + 1) % NUM_COLS == 0)
         cout << endl;
   }
}

void hash_tables::insert(char* s)
// Inserts string <s> in hash table, provided
// no collision is found. This function must be
// modified to handle collisions.
{
   if (! contains(s))
   {
      int h = slot(s);
      if (element[h] == NULL)
      {
         element[h] = new char[strlen(s)+1];
         strcpy(element[h],s);
      }
      else
         ++num_collisions;
   }
}

bool hash_tables::contains(char* s) const
// Tells whether string <s> is in the
// hash table.
{
   int h = slot(s);
   return (element[h] != NULL &&
         ! strcmp(s,element[h]));
}

int hash_tables::hash(char* s) const
// Hash function, adapted for this table.
// See alternative hash function,
// <mid_square>, below.
{
   return sum_of_letters(s) % SIZE;
}

int hash_tables::slot(char* s) const
// Finds slot in table where key <s> should go
// or already is. If value returned is
// subscript of a full slot,
// <s> was found there.
// Otherwise, <s> could be inserted there.
// To handle collisions, this function must be
// modified to do a linear probe in case of a
// collision.
{
   return hash(s);
}

int hash_tables::get_collisions() const
// Access function.
{
   return num_collisions;
}

// General-purpose hash functions:

int sum_of_letters(char* s)
// Returns sum of ASCII values of letters
// in <s>.
{
   int sum = 0;
   for (int i=0; i < (int)strlen(s); ++i)
      sum += s[i];
   return sum;
}

int mid_square(char* s)
// Returns mid-square of 8-character
// string <s>.
{
   int product[8] = {0,0,0,0,0,0,0,0};
   for (int i=7; i >= 0; --i)
      for (int j=7; j >=0; --j)
         product[i] += s[i] * s[j];
   return (product[4] << 8) + product[5];
}
```

Brief introduction to automata

- Automata: simple models of computation that represent imaginary machines
- Describe recognizers of certain sets of sequences of inputs, and transducers that map input sequences to output sequences
- The machine moves among its *states* upon reading inputs
- Intuitive example: a television is in one of these states: {off, channel 1, channel 2, ... channel 69}
- Deterministic finite automata (DFAs) are the simplest model in a hierarchy of increasing expressiveness
- DFA $M = (\Sigma, Q, \delta, q_0, F)$, where
  Σ is the input alphabet;
  Q is the set of states of M;
  δ is a table specifying the state into which M goes from a given state on a given input
  q_0 is the state in whicm M starts;
  F is a set of states such that, if a certain input string leaves M in a state in F, then we say M "accepts" that input string
- *Example DFA:*
  $\Sigma = \{flip\}$
  $Q = \{ off, on \}$
  $\delta(off, flip) = on$ and $\delta(on, flip) = off$
  $q_0 = off$
  $F = \{ off \}$
  Problem: what sequences does the above DFA accept?

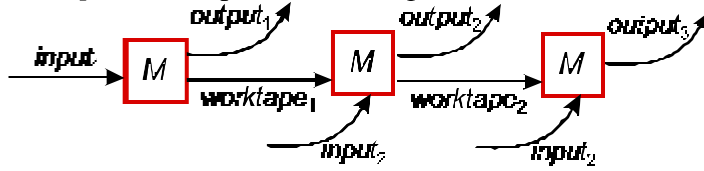


- *Problem:* Diagram the DFA that is defined by:
  $\Sigma = \{0, 1\}$
  $Q = \{ a, b \}$
  $\delta(a, 0) = a; \delta(b, 0) = a; \delta(a, 1) = b; \delta(b, 1) = b$
  $q_0 = a$
  $F \{ b \}$
  What input strings does this DFA accept?
- Extension of DFAs: Add outputs to transition function δ (finite transducers)
- Any machine with finite storage capacity and I/O may be described as a DFA or finite transducer
- Further extensions:
  Pushdown automata (add stack add push and pop operations to δ)
  Turing machines (add tape; add tape-writing operations and left/right moves to δ)
- Any computable function may be computed by a Turing machine (every TM corresponds to an algorithm)



- Any stored-program computer (von Neumann architecture) is modeled by a *universal Turing machine* that takes two inputs, a program and an input string, and simulates the activity of Turing machine/program M running with input I, producing output O
- A *sequential interaction machine* (SIM) models a computation in which input alternates with output.

- *Example:* PTM (persistent Turing machine)



Some comments on Shaffer

The textbook often gives quite good explanations. It also sometimes gets lost in details of mathematical calculations of trivial quantities and in details of C++ coding, obscuring some of the main points.

The emphasis on mathematical techniques is a plus, but in your reading you may wish to concentrate on the main mathematical principles involved, rather than the smaller details of equations and simplifications. Unless you are planning to use templates in your homework, you may wish to skim over long discussions of coding, concentrating on the coding of algorithms rather than the C++ issues of classes and templates.

The chapters and sections listed below are optional reading, covering material we will not cover in our course. You can mark these in your book to skip if necessary. Reading the whole book is of course useful. If you skip the optional sections and skim the math and coding details, you will have only 200 or 300 pages to read at most. Over fifteen weeks, this only amounts to 20 pages a week, a minimal amount of reading for a college course. Do it!

Optional chapters (single digit) and sections (with dot):

2.6.1
3.9
5.6
6.2
6.1.1
6.1.2
6.3.1
6.5
7.3
7.5
7.7
8
9.2
9.3
10
12
13
14.3
15