CP2 Revision

theme: file access and unix programs

file access in C

basic access functionality:

- **`FILE *fopen(const char *filename, const char *mode);`**
  This function returns a pointer to a file stream (*or **NULL** if it fails*) and takes two strings as parameters. The first of these (*filename*) is the name of the file that is to be opened and the second one (*mode*) determines the access mode which is to be used for manipulating the file.

- **`int fclose(FILE *filepointer);`**
  This function is used to close already open files – it returns **1** if file was closed or **0** if the operation failed.

common mode switches for the *`fopen()`* function:

`"r"`	*read*	open a text file (*ASCII*) for reading
`"w"`	*write*	open a text file (*ASCII*) for writing
`"a"`	*append*	open a text file (*ASCII*) for appending
`"rb"`	*read binary*	open a binary file for reading
`"wb"`	*write binary*	open a binary file for writing
`"ab"`	*append binary*	open a binary file for appending

sequential file access

- **`int fgetc(FILE *file);`**
  returns the next character from the opened *file* as an unsigned char; if an error occurs or the end of the file is reached the EOF value is returned.

- **`char *fgets(char *string,int length,FILE *file);`**
  returns the next "*length*-1" characters from the opened *file* into the *string*, terminating it with `\0`; if the end of the line is reached before "*length*-1" characters have been read, fgets terminates the string; if an error occurs or the end of the file is reached NULL is returned.

- **`int fscanf(FILE *file,char* formatstring,...);`**
  does a scanf of the next line from the *file*.

- **`int fputc(int character,FILE *file);`**
  writes the *character* into the *file*; if an error occurs EOF is returned.

- **`int fputs(const char *string,FILE *file);`**
  writes the *string* into the *file*; if an error occurs EOF is returned.

- **`int fprintf(FILE *file,char* formatstring,...);`**
  does a printf into the *file*.

random file access

- **`size_t fread(void *buffer,size_t bytes,size_t quantity, FILE *file);`**
  reads up to ***quantity*** elements of size ***bytes*** from file ***file*** into the memory associated with ***buffer***. Returns the actual number of read elements (*which can be less than quantity*). ***`feof`*** and ***`ferror`*** can be used to determine a cause for 'missing' elements. If ***0*** elements are read in, the *buffer* remains unchanged.
- **`size_t fwrite(const void *buffer,size_t bytes,size_t quantity,`**
  `FILE *file);`
  will write ***quantity*** elements of size ***bytes*** starting from memory address ***buffer*** into the ***file*** (*will position the read/write pointer to the end of the written block*). Returns the number of successfully written elements. If an error occurred, that value will be smaller than *quantity*.
- **`int fgetpos(FILE *file,fpos_t *pos);`**
  finds the current read/write pointer position in ***file*** and copies its value into the variable that ***pos*** points to. Returns ***0*** if successful.
- **`int fseek(FILE *file,long offset,int position);`**
  is used for repositioning the read/write pointer within the ***file***. It will be positioned ***offset*** bytes from the given ***position***, which can be defined using one of the three macros

`SEEK_SET`	*start of file*
`SEEK_CUR`	*current position of read/write pointer*
`SEEK_END`	*end of file*

  Returns ***0*** if successful.

random file access (2)

- **`long`** ***`ftell`*****`(FILE *file);`**

  returns the number of bytes from the start of the ***file*** until the current read/write pointer position. Returns ***-1*** if it fails.

- **`int`** ***`fsetpos`*****`(FILE *file,const fpos_t *pos);`**

  is used for positioning the read/write pointer within the ***file***. It will be positioned at the position contained in the variable that ***pos*** points to (*the positional value would be of the type used by the fgetpos function*). Returns ***0*** if successful.

- **`void`** ***`rewind`*****`(FILE *file);`**

  resets any error and end-of-file marker for the file stream associated with ***file*** and sets the read/write pointer to the start of the file. In effect this operation is identical to:

  ```
  fseek(file,0,SEEK_SET);
  clearerr(file);
  ```

command-line arguments in C

- like all programs, C programs can receive command line arguments from the shell that calls the program
- in a C program these arguments are passed into the program through parameters in the program's ***main*** function

In the **ANSI C** standard this is done using ***2*** parameters:

1. an integer value which contains the number of arguments - by definition this argument has the name ***argc*** (*argument counter*).
   Note: *there is at least one argument in every program, that argument being the name of the program itself*
2. an array of character strings each of which contains one of the arguments - by definition this argument has the name ***argv*** (*argument vector*).
   the name of the program itself is always stored in **argv[0]** .

Syntax: **`int/void main(int argc,char *argv[])`**

command-line arguments (2)

- command line arguments are usually interpreted at the start of a program (*variables that are used as flags for options are set*).
- this is often done as a **large switch statement** which either evaluates the argument counter (*to allow access to a specific argument*) or which evaluates the first character of an argument (*to determine what argument it is*) within a loop stepping through all arguments.

example:

a simple C program which prints out and numbers all command-line arguments that are passed into it from the shell.

```
#include<stdio.h>
void main(int argc,char *argv[])
{
  int i;
  for(i=0;i<argc;i++)
    printf("argument %d is %s\n",i,argv[i]);
}
```

recap:

- passed into a C program through two parameters in the program's ***main*** function:
  `int` **`argc`** (*argument counter*)
  `char` **`*argv[]`** (*argument vector*)
- there is at least one argument in every program, that argument being the name of the program itself which is always stored in **argv[0]**

Syntax: **`int/void main(int argc,char *argv[])`**

case studies: unix programs

Unix filter programs

- programs that read data (*from a file or the standard input*), process it (*manipulation of one form or another*) and produce some resulting data output (*written into a file or the standard output*)

- <u>examples:</u>

`grep`	- search a data stream for various keywords
`sed`	- stream-editor manipulates/modifies data streams
`cat`	- concatenates data streams
`wc`	- wordcount: word- character- and line-counting

- <u>other unix programs:</u>

`echo`	- prints ist input onto the standard output
`touch`	- touches (*updates*) files

example: **echo**

a) print all output from the command-line arguments:

```
#include<stdio.h>
void main(int argc,char *argv[])
{
  int i;
  for(i=1;i<argc;i++)
      printf("%s ",argv[i]);
  printf("\n");
}
```

b) more complex - suppress newline if -n argument found:

```
#include<stdio.h>
void main(int argc,char *argv[])
{
  int i,n=1;
  for(i=1;i<argc;i++)
  {
      if(strcmp(argv[i],”-n”)==0) /* if -n argument is found */
         n=0;  /* set newline-flag to 0 */
      else
         printf("%s ",argv[i]);
  }
  if(n==1) printf("\n"); /* print newline if newline-flag is 1 */
}
```

example: **grep**

grep finds character sequences in files. As command line arguments ***grep*** takes a string followed by a list of files to search in. If the string is found in a file, ***grep*** prints the filename, the line number and the line with the string.

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
/* my grep program */
int main(int argc, char *argv[])
{
  FILE *f;          /* the current file pointer */
  int i,lineNr;     /* file index and line number */
  char line[256];   /* line of text */
  if(argc<3)        /* if too few arguments */
  {
    printf("not enough parameters!\n");
    exit(EXIT_FAILURE); /* print error message and quit */
  }
  for(i=2;i<argc;i++)   /* else cycle through all listed files */
  {
    f=fopen(argv[i],"r"); /* open file for reading */
```

example: **grep** (*cont.*)

```
    if(f==NULL)    /* if unable to open print error and ...*/
    {
      printf("Unable to open file %s\n",argv[i]);
      continue;    /* ... step on to the next file in loop */
    }
    else
    {
      for(lineNr=1;!feof(f);lineNr++)   /* cycle through lines */
      {
        fgets(line,256,f);              /* retrieve next line */
        if(strstr(line,argv[1])!=NULL)  /* search line for search-pattern */
          printf("%s:%d:%s\n",argv[i],lineNr,line);
                                        /* print result if successful*/
      }
      fclose(f);   /* close file */
      f=NULL;
    }
  }
  printf("\n");
  exit(EXIT_SUCCESS);
}
```

example: a simplified **cat**

```
#include<stdio.h>

int main(int argc,char *argv[]) /* a simple cat */
{
  FILE *f;              /* current file pointer */
  int c,i;              /* character and file conter */
  f=stdin;              /* first read from the standard input */
  while((c=fgetc(f))!=EOF) printf("%c",c);
  printf("\n");         /* a newline after every file */
  for(i=1;i<argc;i++)   /* cycle through list of files */
  {
    f=fopen(argv[i],"r"); /* open */
    if(f!=NULL)           /* if successful then read characters */
    {
      while((c=fgetc(f))!=EOF) printf("%c",c);
      printf("\n");
      fclose(f);
    }
  }
  return 1;
}
```

case study: wc

wordcount filter wc**:** `wc [options] [file(s)]`

wc

reads data directly from ASCII files or from the standard input and counts how many lines, words and/or characters have been read in. Words are separated by white spaces.

wc

accepts three command line options **-l**, **-w** and **-c**. They are used for determining whether just the number of lines (***-l***), the number of words (***-w***) or the number of characters (***-c***) are to be displayed.

(*a full tutorial can be accessed at http://programming.swordfighter.co.uk*)

simplified wc

```
#include<stdlib.h>      /* simple wordcount */
#include<stdio.h>
void main(int argc,char *argv[])
{
  char letter;
  int e,words=0,lines=0,chars=0;
  if(argc!=1)
  {
    perror("WRONG NUMBER OF PARAMETERS\n"); exit(EXIT_FAILURE);
  }
  while((letter=fgetc(stdin))!=EOF)
  {
    if(lines==0)                                /* find start of first word */
    {
      if(letter==' ' || letter=='\t') e=0;
      else e=1;
    }
    chars++;                                    /* increase character count */
    if(letter=='\n') lines++;
    if(e)
    {
      if(letter==' ' || letter=='\t')
      {
        e=0;
        words++;
      }
    }
    else if(letter!=' ' && letter!='\t') e=1;
  }
  printf("\t%d \t%d \t%d\n",lines,words,chars);
}
```

a more complex wc

Additions to the program that are necessary for evaluating optional command-line arguments:

```
int w=0,c=0,l=0,i,j,words,lines,chars;
...
if(argc>1)
  for(i=1;argv[i][0]=='-';i++)
    for(j=1;argv[i][j]!='\0';j++)
    {
      switch(argv[i][j])
      {
        case 'w':w=1; /* w found - set w to true */
                 break;
        case 'c':c=1; /* c found - set c to true */
                 break;
        case 'l':l=1; /* l found - set l to true */
                 break;
        default: printf("UNKNOWN OPTION\n");
                 exit(EXIT_FAILURE);
      }
    }
```

After evaluating the optional command-line arguments all following arguments can be assumed to be filenames.

case study: file access and dynamic data structures

combining file access with dynamic datastructures

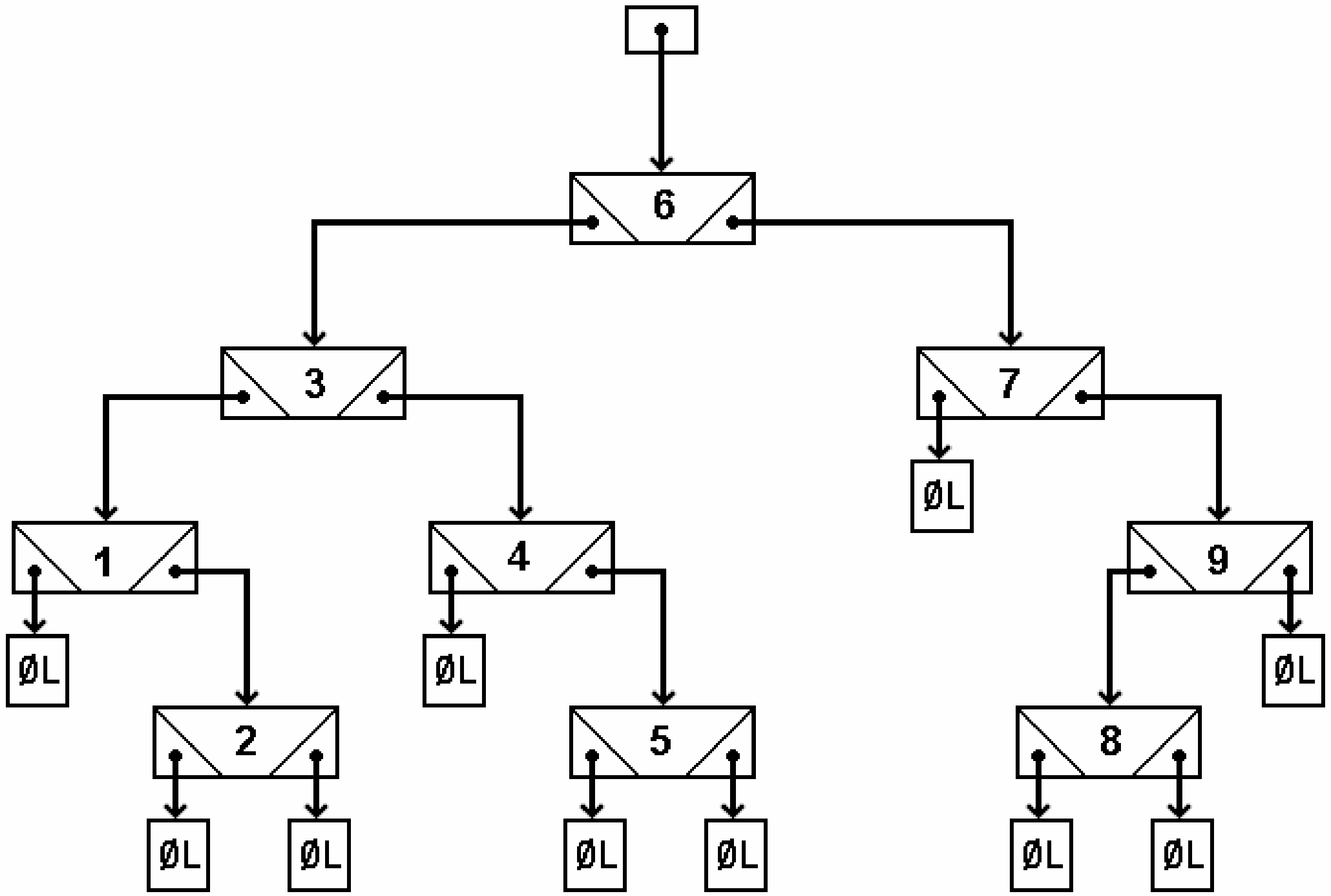
questions to ask:

what would be the requirements in regard to datatypes used and functions employed, necessary for saving a binary tree to a file?

how would the algorithm that saves the binary tree out into a file work if it should also be able to reload that tree into memory, keeping the original tree structure intact?

how would the algorithm for loading the tree into memory work?

a binary tree



saving a tree to a file

- nodes cannot be saved directly as the dynamic links cannot be guaranteed to be available when reloading…

  solution: *an intermediate datatype*

  ```
  typedef struct
  {
     char left;
     char right;
     int  data;
  }  fileEntry;
  ```

- left and right members of the ***fileEntry*** are flags that mark if there is a branch below the node (*value 1*) or if there is no branch (*value 0*)

- since the tree is going to have to be rebuilt from its root when the saved tree is reloaded, the root node has to be saved first:
  the algorithm for saving the tree is pre-order recursive

sample function for saving

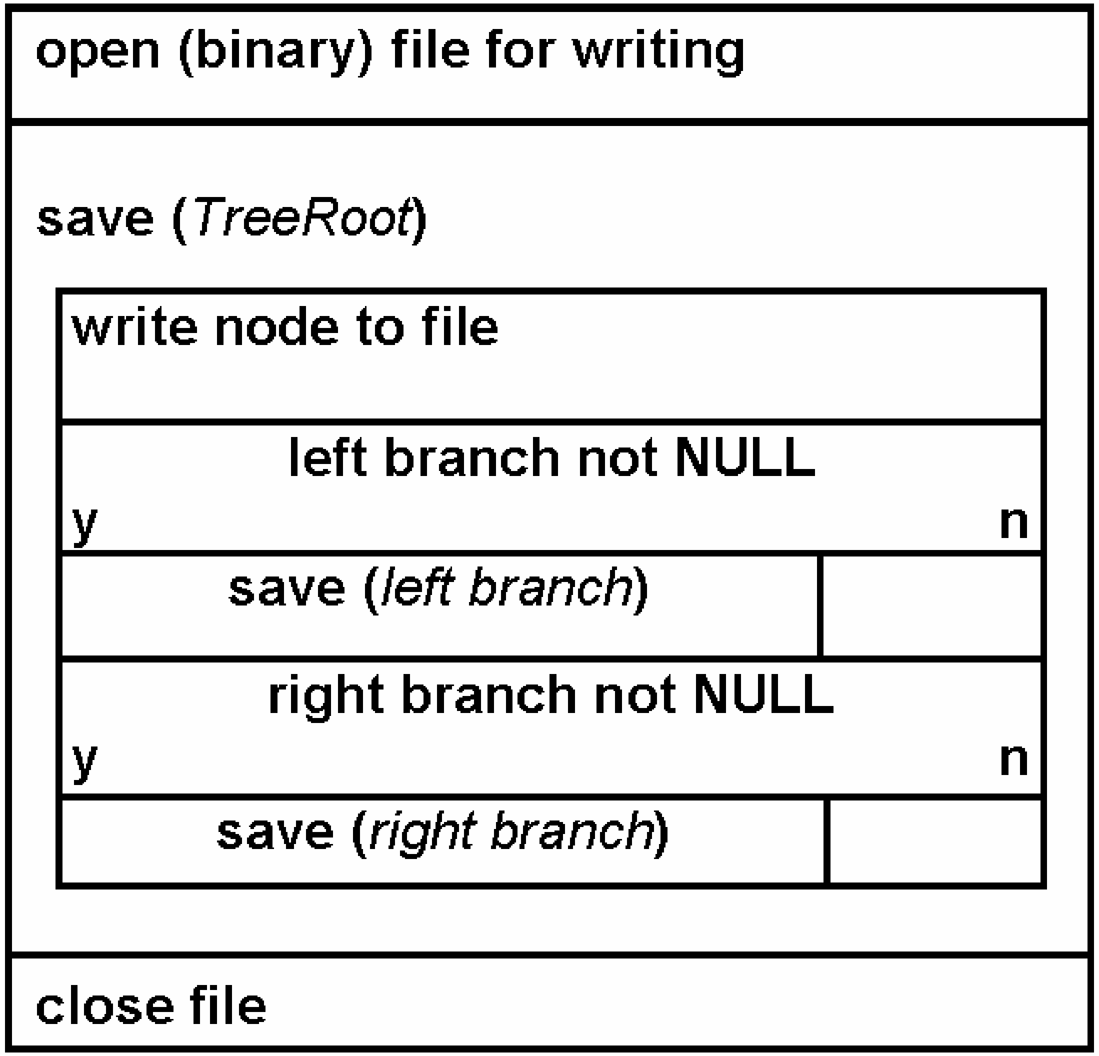
```
int save(nodePtr root,FILE *outfile)
{
  fileEntry entry;
  if(root==NULL)  return 0;
  entry.data=root->data;
  if(root->left!=NULL)
    entry.left=1;
  else
    entry.left=0;
  if(root->right!=NULL)
    entry.right=1;
  else
    entry.right=0;
  fwrite(&entry,sizeof(fileEntry),1,outfile);
  save(root->left,outfile);
  save(root->right,outfile);
  return 1;
}
```

algorithm for saving the tree

1. copy current node into ***fileEntry***

2. store value 1 for branches that exist and value 0 for branches that do not exist within the ***fileEntry***

3. write ***fileEntry*** to file and traverse the tree pre-order to recursively save it to file



```
open (binary) file for writing
save (TreeRoot)
    write node to file
    left branch not NULL
        y: save (left branch)
        n:
    right branch not NULL
        y: save (right branch)
        n:
close file
```

loading a tree from a file

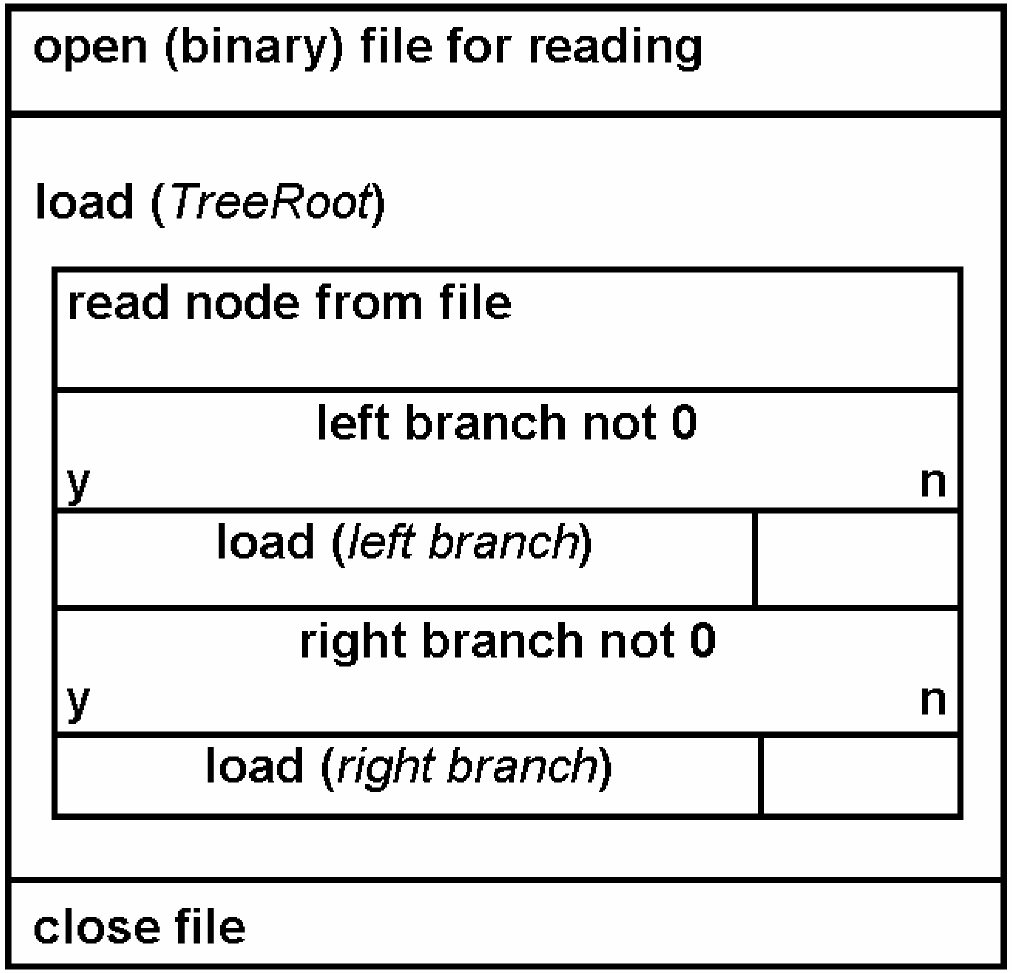
- dynamic links are not saved, but must be generated through memory allocation while the tree is reloaded

- left and right members of the ***fileEntry*** are flags that mark if there is a branch below the node which will direct the traversal path during the rebuilding process of the tree

algorithm for loading the tree

1. copy current ***fileEntry*** into a newly allocated tree node

2. rebuild tree pre-order by traversing into branches that are marked as existing within the ***fileEntry***



sample function for loading

```
int load(nodePtr *root,FILE *infile)
{
  fileEntry entry;
  nodePtr temp;
  fread(&entry,sizeof(fileEntry),1,infile);
  if((temp=(nodePtr)malloc(sizeof(node)))!=NULL)
  {
    temp->left=NULL;
    temp->right=NULL;
    temp->data=entry.data;
    *root=temp;
  }
  else  return 0;
  if(entry.left==1)
    if(load(&(temp->left),infile)==0)  return 0;
  if(entry.right==1)
    if(load(&(temp->right),infile)==0) return 0;
  return 1;
}
```