CP2 Revision

theme: dynamic datatypes & data structures

structs

- can hold any combination of datatypes
- handled as single entity

```
struct <identifier>
{
    <datatype> <member identifier>;
    <datatype> <member identifier>;
    …
};
```

structs (*2*)

- when declared, struct keyword must be used

```
struct  myStruct  myStructVariable;
```

- structure size is at least the size of the sum of all member sizes
- structs can be pre-initialised

self-referential datastructures

- datastructures which can reference a variable of its own kind (*datatype*)
- alternative name:
  „recursive datastructures"
  - has no direct correlation to recursive functions
  - however can be effectively used with recursive functions
- implemented using structures that have a pointer to their own datatype as a member

self-referential structs

```
struct selfRef
{
   struct selfRef *reference;
};
```

Dynamic Memory Allocation

- during program **run-time** memory from the **heap** of the system RAM can be allocated (*reserved*)
- allows programs to dynamically handle data for which exact memory size requirements were unknown at **compile-time**
- functions for dynamic memory allocation are provided by ANSI C standard library (*header file **stdlib.h** must be included*)

3 functions for DMA

- all DMA functions return void pointer to allocated memory or **NULL** if allocation fails
- malloc
  - simplest DMA function
  - takes 1 parameter (*wanted size in bytes*)
- calloc
  - for allocating arrays of same datatype
  - takes 2 parameters: quantity and element size (*in bytes*)
  - pre-initialises all elements to 0 (or NULL if pointers)
- realloc
  - for changing size of dynamically allocated memory blocks
  - takes 2 parameters: pointer to allocated memory and new required size (*in bytes*)
  - if pointer (*1st parameter*) is NULL, realloc acts like malloc

notes for using DMA

- memory that has been allocated during **run-time** must be freed during **run-time**
  - failure to do so results in memory leaks
  - use function free (*takes pointer to allocated memory as parameter*)
- allocated memory may have to be (type-)cast to the target datatype to prevent compiler warnings

  Example:

```
int *array=(int*)malloc(4*sizeof(int));
/* generate an integer array with 4 elements */
```

applying DMA – dynamic arrays

When would it be useful to use a dynamic array?

- dynamically allocated during run-time
- can be used just like any other array
- elements can be accessed using the index operator **[** *i* **]**
- data is held within a single continuous block in memory

problems with arrays

- if too big it may be impossible to allocate large enough continuous blocks of memory
- dynamically resizing arrays can take a long time
- array elements can only (*really*) be added at or removed from end of array

linked lists

needed:

- a dynamic datastructure which can grow & shrink at run-time
- elements to the datastructure insertable and deletable at any position within the structure

linked lists are:

- dynamically created during program run-time
- sequential collection of self-referential elements (*called* ***nodes***)
- elements are accessed linearly by sequentially traversing the list from start to finish

linked lists (*2*)

What is the conceptual difference to a dynamic array?

- data not in single block of memory but each node separately with link to next node
- elements cannot be accessed randomly but must be accessed sequencially
- slower access than arrays but data can be added/removed from anywhere within the list

linked lists - review

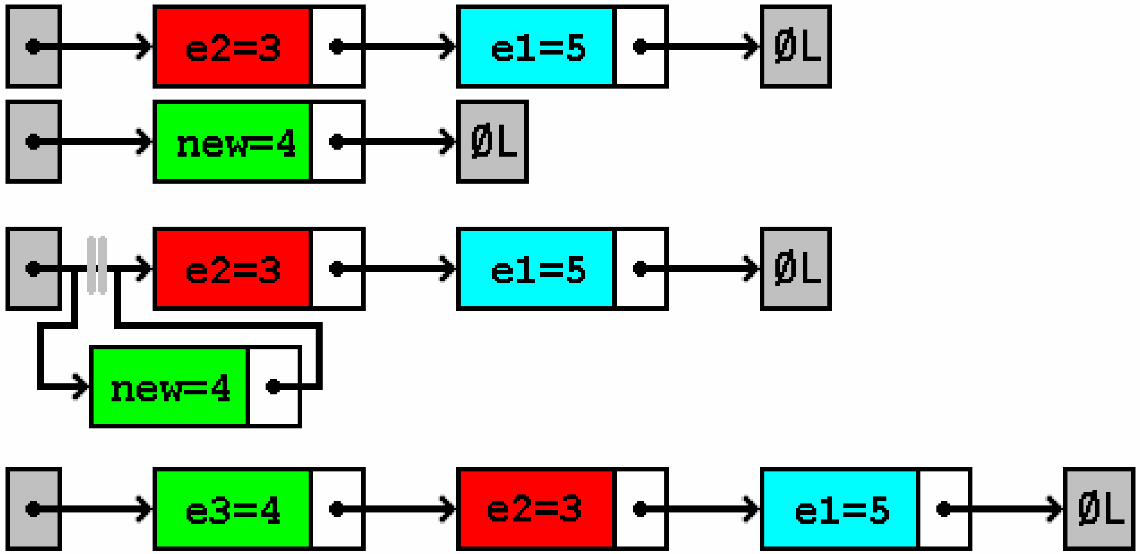
- singly linked list
  - **one directional sequential access**
- stacks
  - **push : insert at head**
  - **pop : remove from head**
- queues
  - **enqueue : insert at tail**
  - **dequeue : remove from head**
- circular buffers
  - **two base pointers (***read/write***)**
  - **no beginning and no end (***predefined number of nodes***)**
- doubly linked lists
  - **bi-directional sequential access (***2 pointers per node***)**

singly linked list

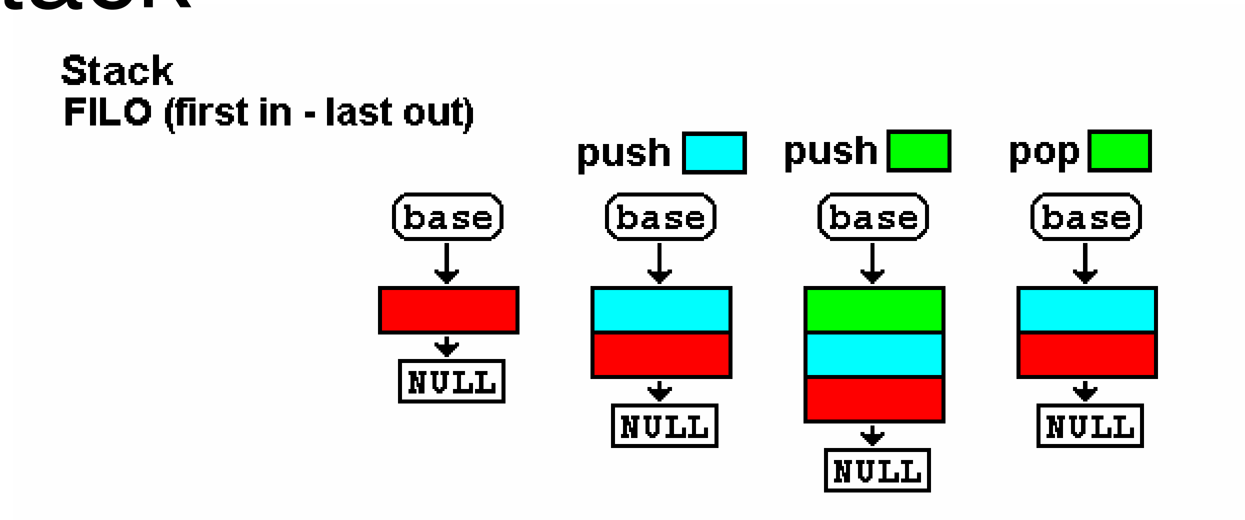
```
typedef node* nodePtr;

int insert(nodePtr*,int);
```

inserting at head of list

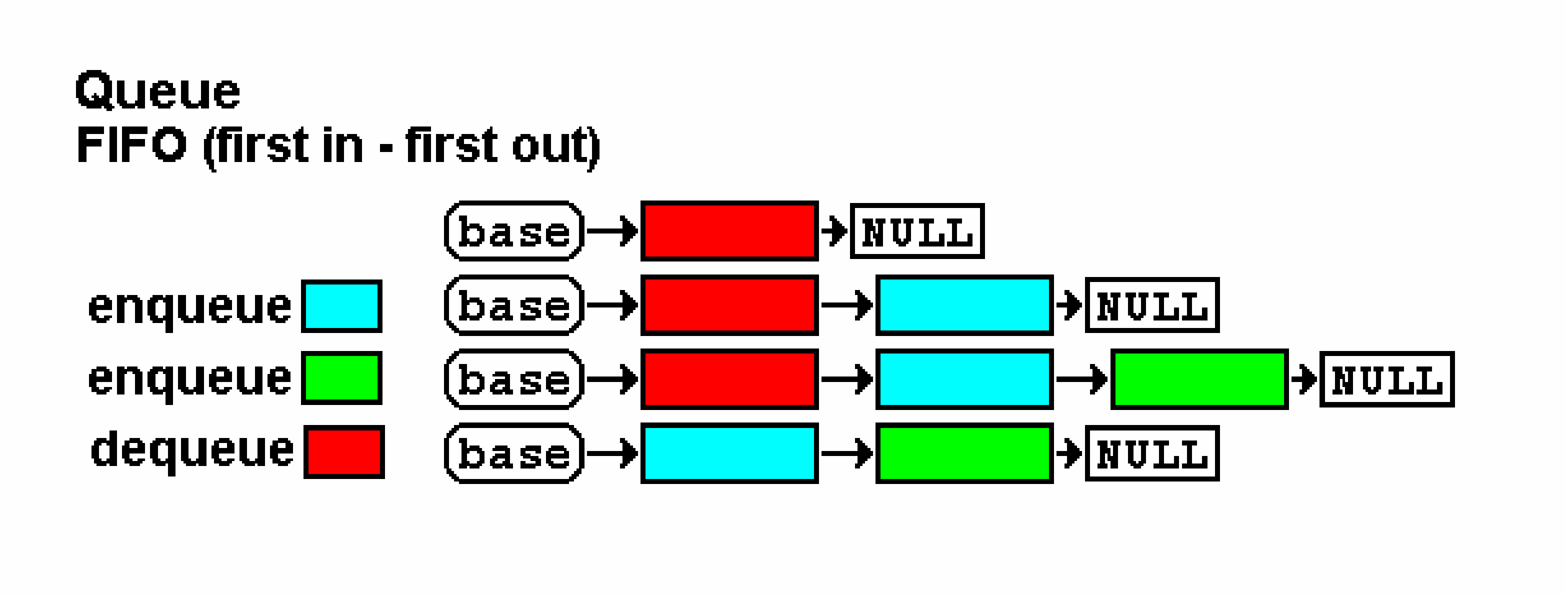


stack



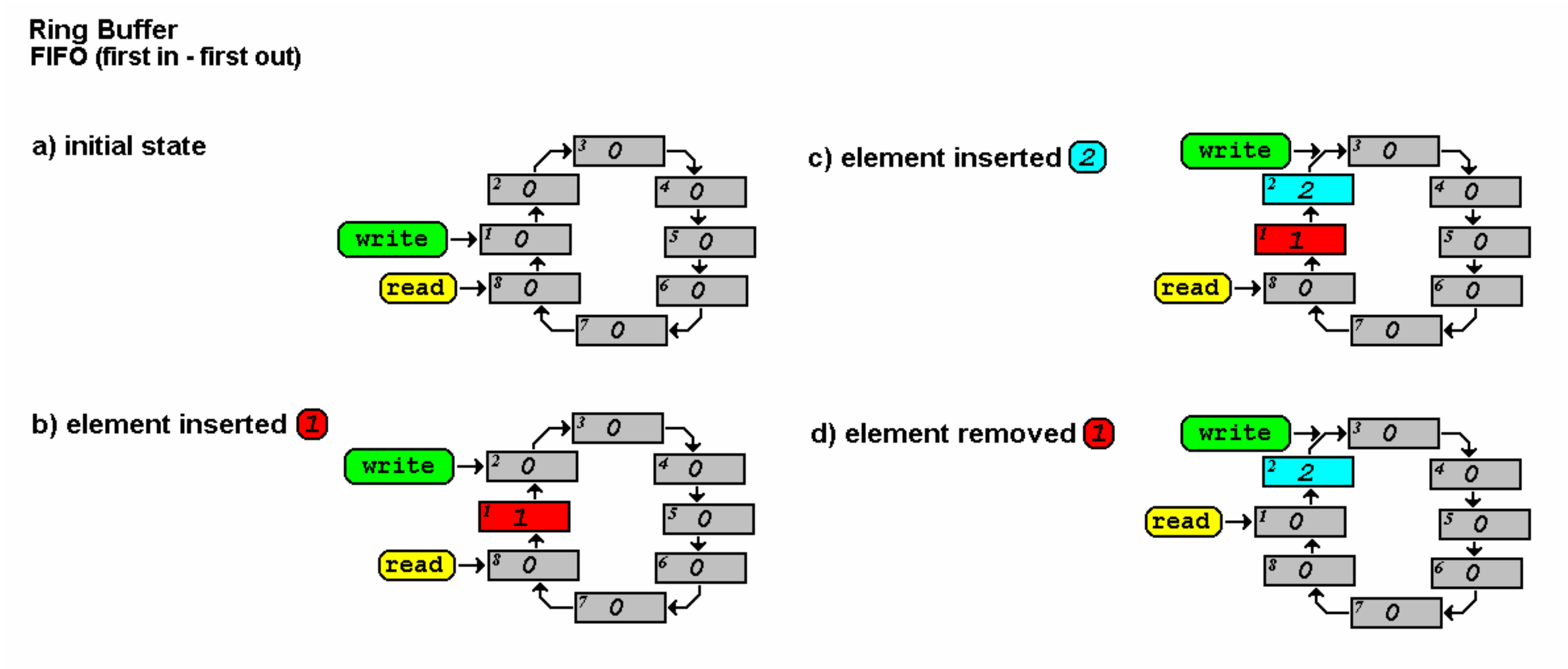
- **push** - list insert at head
- **pop** - element removal at head
- **top** - pop & push

queues



- **enqueue** - identical to list insert at tail
- **dequeue** - identical to stack's pop

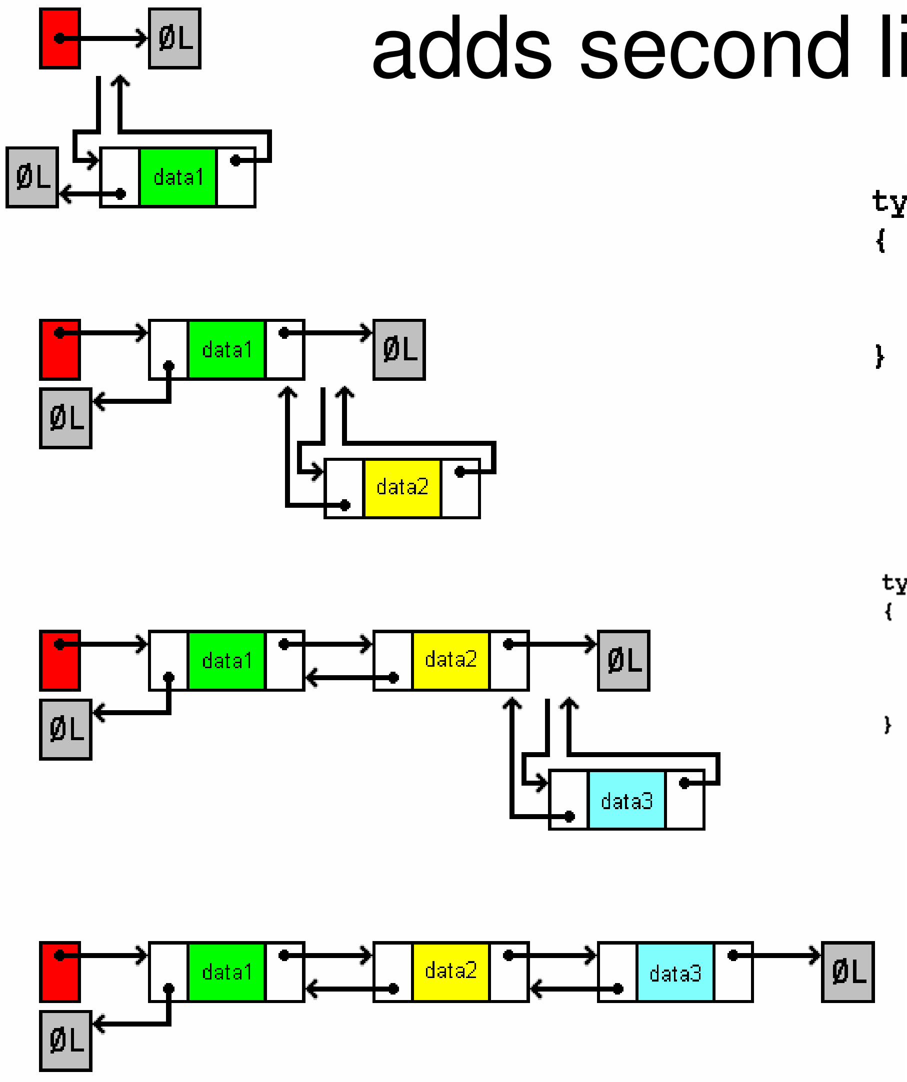
circular buffers



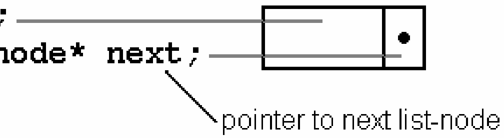
- **writing**: enter data & advance write pointer
- **reading**: advance read pointer & retrieve data

doubly linked list

adds second link to singly linked list!

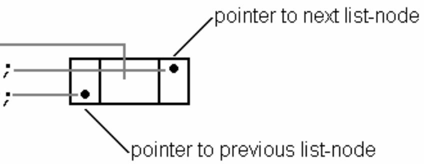



```
typedef struct _node
{
  int data;
  struct _node* next;
} node;
```



Doubly Linked List

```
typedef struct _node
{
  int data;
  struct _node *next;
  struct _node *prev;
} node;
```



problems with singly linked lists

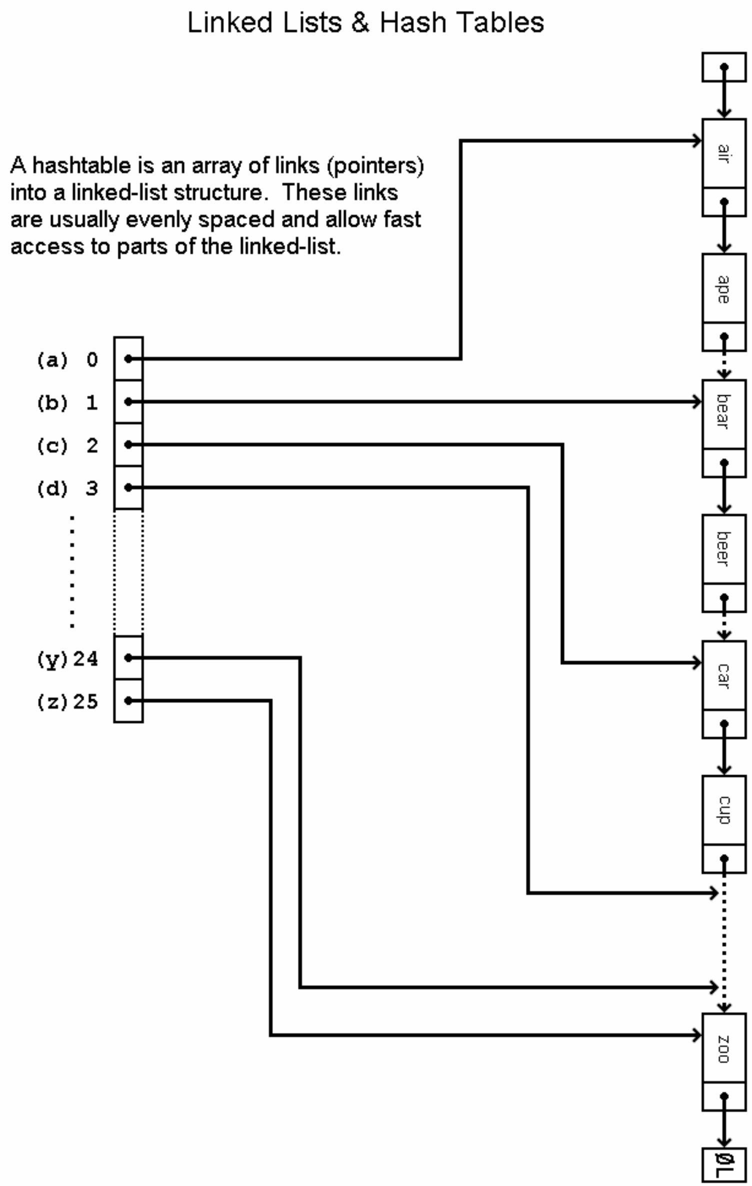
- only one-directional (*linear*) sequential access
- worst case (*of necessary*) steps for finding element is no. of list nodes in list ***n***

problems with doubly linked lists

- only bi-directional sequential access
- worst case (*of necessary*) steps for finding element is no. of list nodes in list ***n***
- average case (*of necessary*) steps for finding element is half the no. of list nodes in list ***n/2***

the answer: hash tables

- combination of linked list & dynamic array:
  - array of links (*pointers*) into a linked list structure
- method to optimise performance of large linked lists
- links usually spaced evenly to allow fast access to parts of linked list
- reduces some disadvantages of linear datastructures



- procedure of generating a hash table is called ***hashing***
- simplest form would be to store all elements with same characteristics in a sub-list
- hashing function generates ***key*** (*correct position in hash table*) from analysing data element
- this key generation can be quite complex if table is to be balanced
- if amount of data changes so that hash table becomes unbalanced, the necessity to recalculate the hash table may arise.
- recalculating the hash table is called ***rehashing***

problems with linked lists

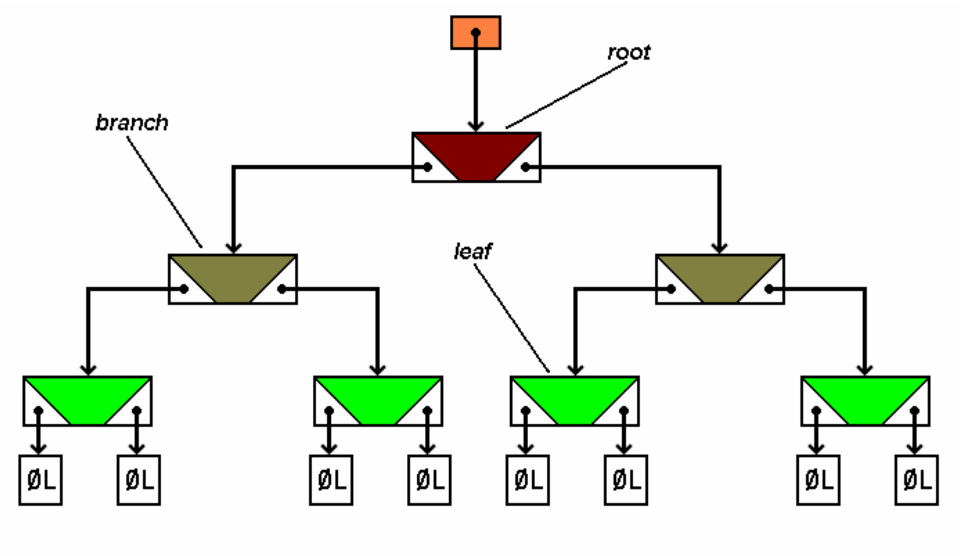
- only sequential access

solution: tree structures

- non-linear self-referential dynamic data structures
- data elements are also called nodes
- tree nodes each contain two or more links

another look at tree structures

- nodes referred to as ***branches*** or ***leaves*** (*if they do not branch off into further nodes*)
- base of a tree is called ***root***
- a node / sub-tree below the root is called ***child*** node
- The level of sub-branching in a tree is called the **depth** (*or height*) of the tree.
- nodes at the same depth (*below the root*) are called ***siblings***

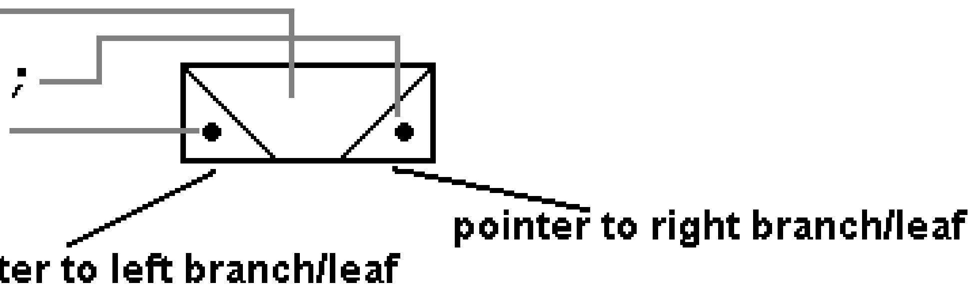


binary trees

- each node contains links to up to 2 nodes
- all nodes except the root have exactly one predecessor

binary = 2 branches/leaves per node

```
typedef struct _node
{
  int data;
  struct _node *right;
  struct _node *left;
} node;
```



binary trees - traversal

3 recursive algorithms for traversing a binary tree:

InOrder

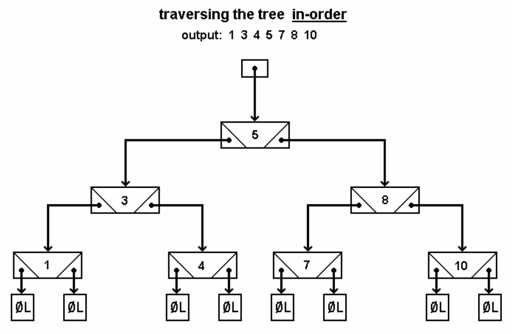
- traversing the tree in-order means that the left branch of a node is processed first before the node itself and then finally the right branch;

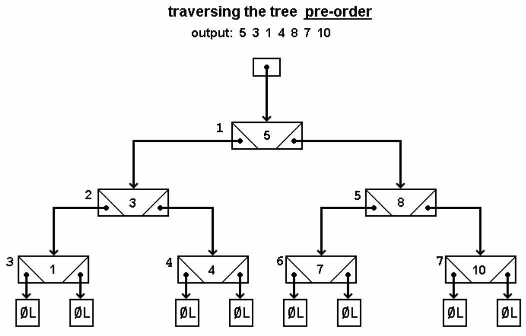
PreOrder

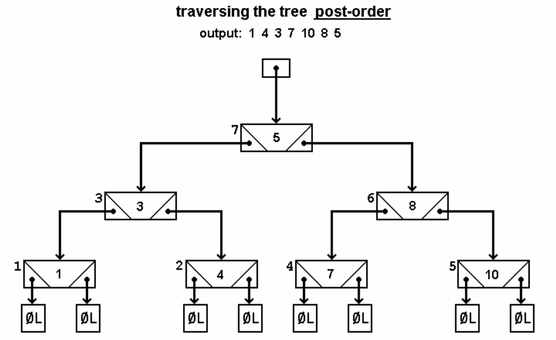
- traversing the tree pre-order means that a node itself is processed first before the left branch and then finally the right branch;

PostOrder

- traversing the tree post-order means that the left branch of a node is processed first before the right branch and then finally the node itself;



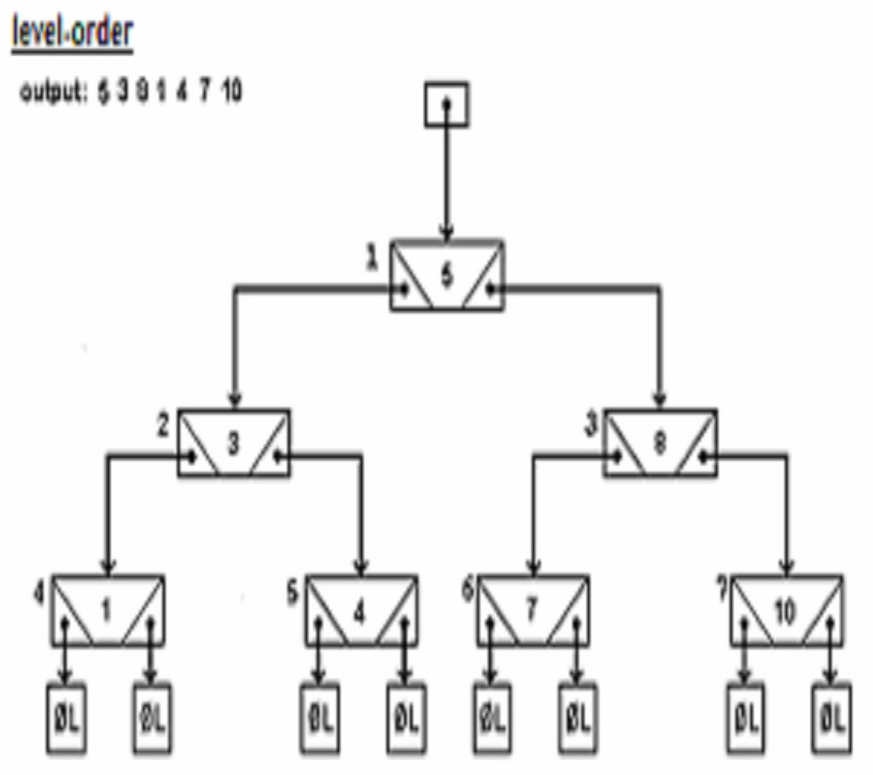




binary trees – breadth first traversal

- Breadth-first traversal of a tree means that each level (*of depth*) of the tree is searched one after the other.
- The algorithm is called “level-order” traversal:

1. root is entered into a queue of current level (*of depth*) nodes.
2. queue is emptied and nodes are processed as they exit the queue and all child nodes are added to a queue of the next level of depth.
3. when the queue of the current depth is empty it is replaced with the queue of the next level of depth.
4. steps 2 and 3 are repeated until both queues (*current level of depth and next level of depth*) are empty.



binary trees - balancing

balanced (*access optimised*) trees

- a tree with same number of nodes in each branch is called a *balanced tree*
- if depth of each branch is also the same, it is called *height-balanced*
- balancing a tree reduces access times
  - worst case: totally unbalanced tree (*all nodes on one side*) – like singly linked list

binary trees – AVL trees

AVL trees

- According to **Adelson-Velskii** & **Landis**, a binary tree is *perfectly* height-balanced if "*the difference in the depth at each branch-node of the tree is 1 or less*".
- By that definition a tree can only be called balanced if the depth of **both** branches of the root node as well as the depth of **both** branches below any other node of the tree is either identical or only differs by one.
- Having these so-called AVL-trees reduces the average access times for the nodes in the tree: in a perfectly balanced binary tree containing 1000 elements (*about 2^10 elements, i.e. depth 10*), retrieving a single element will take no more than ***eleven*** comparisons (*best case*). The *worst case* would be a completely unbalanced tree resembling a singly linked list, which might require up to 1000 comparisons in a tree containing 1000 elements.

binary trees – balancing (2)

- if all inner nodes have a number of children that is identical to the tree's out degree, the tree is called a "**full tree**"
- if all leaves of the tree are situated at the same level of depth (*a full tree with a depth difference of 0 between branches*), the tree is called a "**complete tree**"
  - a complete tree is always an AVL 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*

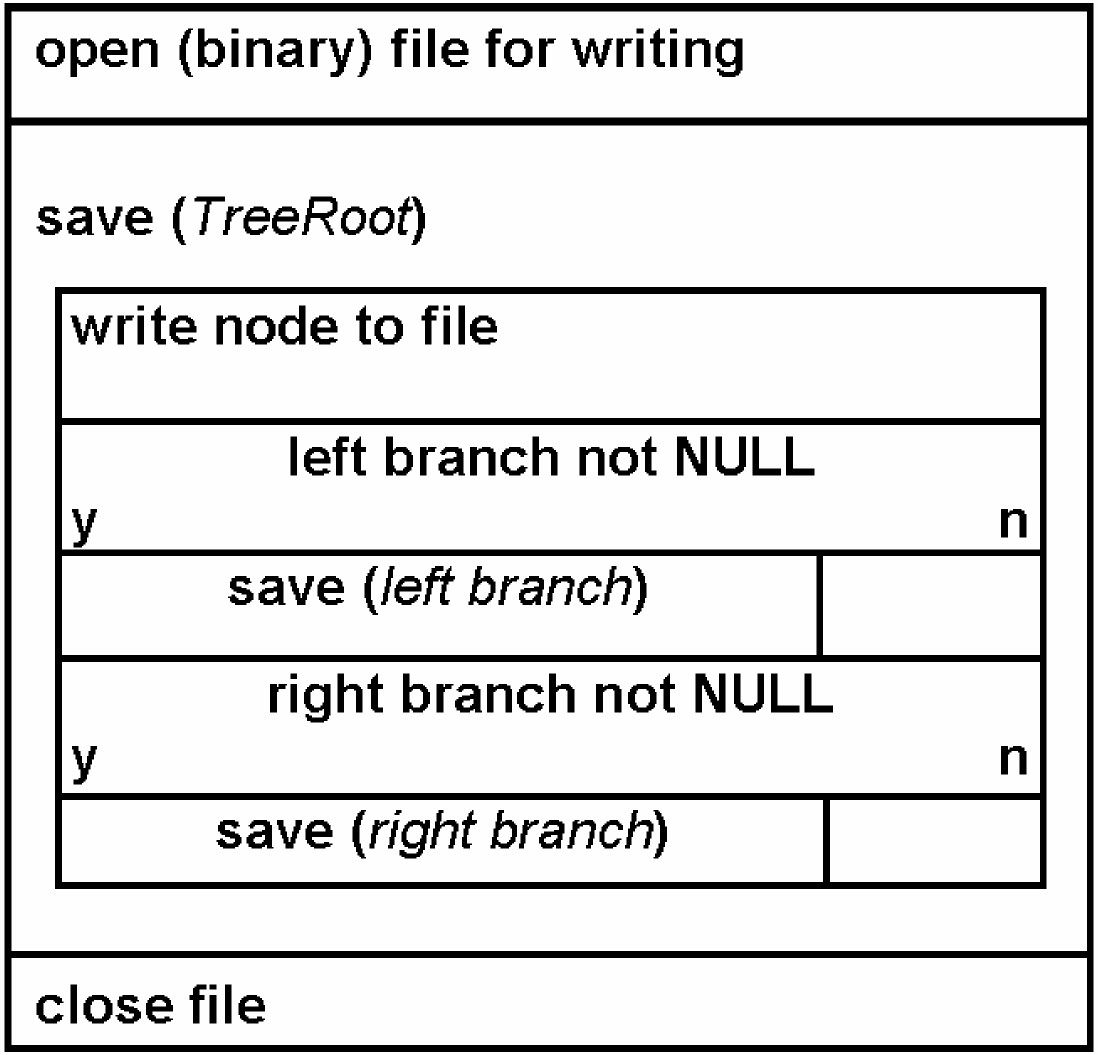
```
typedef struct
{
   char left;
   char right;
   int  data;
}  fileEntry;
```

- 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

algorithm for saving the tree

- copy current node into ***fileEntry***
- store value 1 for branches that exist and value 0 for branches that do not exist within the ***fileEntry***
- write ***fileEntry*** to file and traverse the tree pre-order to recursively save it to file



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;
}
```

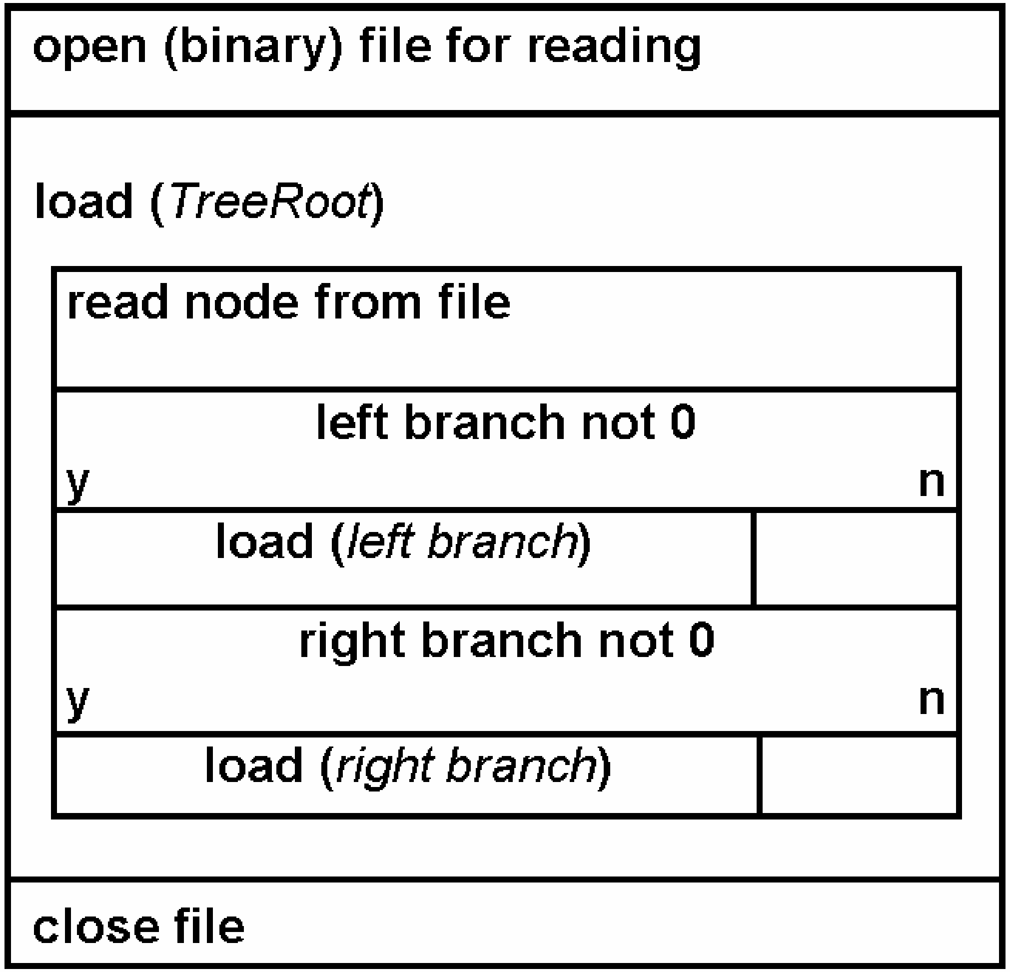
loading a tree from a file

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

- 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

- copy current ***fileEntry*** into a newly allocated tree node
- 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;
}
```