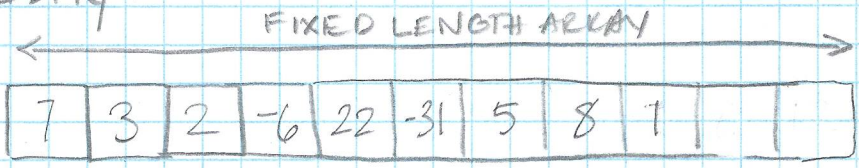
# Data Structures: the singly-linked list

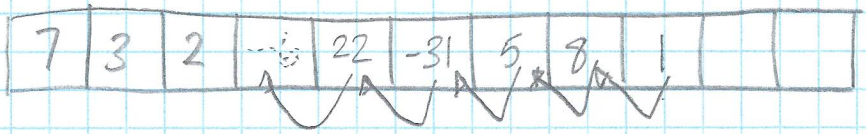


- a dynamic data structure
  ⇒ memory is allocated from the heap at any time, and can be also deallocated

- unlike fixed-length arrays, linked-lists can insert and delete elements easily:



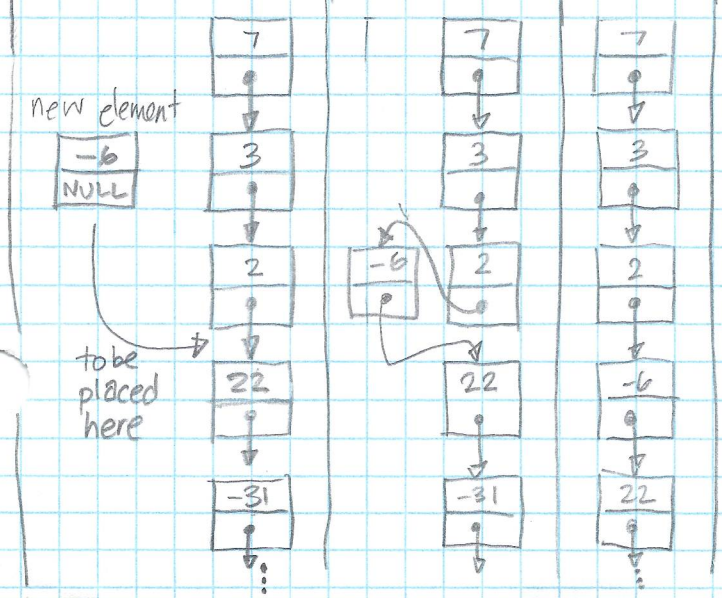

- to delete element "-6" all elements following it must be shuffled one position to the left



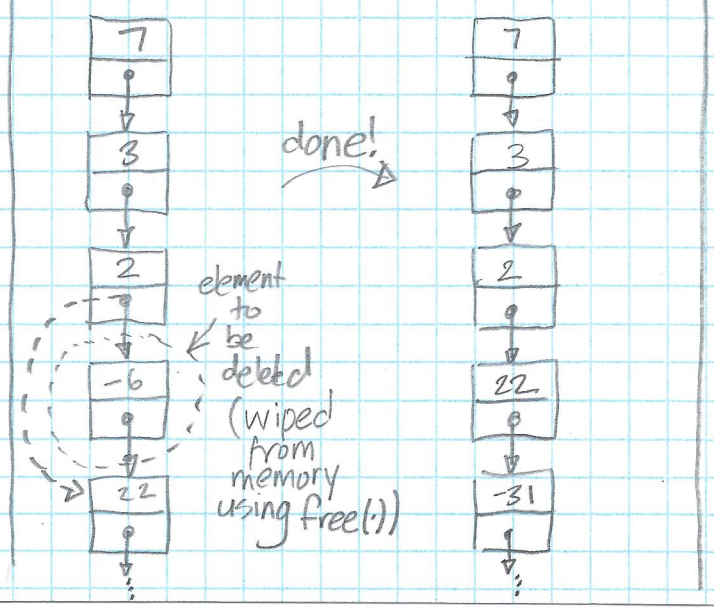

- to insert a new element in the array, many elements may need to shifted to the right to accommodate the new element

- not only are insertion and deletion cumbersome with arrays, but because of their fixed length, the number of possible insertions or additions to the array may be severely constrained

LINKED-LIST INSERTION REQUIRES CHANGING ONLY TWO POINTERS!




LINKED-LIST DELETION IS EASILY ACCOMPLISHED WITH ONE POINTER CHANGE!



- each element of a singly-linked list is also called a node.
- each node consists of data of some kind, and a pointer; the pointer points either to the next node in the list, or, if it is the last element in the list, the pointer is null.
- NULL is a keyword defined in the C standard library, stdlib.h / stdlib.c, used to define null pointers, or pointers which point "nowhere".

EXAMPLE: A NODE DEFINITION IN C

a useful type for the data portion of a node

```
struct data_struct
{
    float X;
    float Y;
    unsigned long int key;
};
typedef struct data_struct data_t;
```

actual definition of node data type

```
struct node_struct
{
    data_t info;
    struct node_struct *pNext;
};
typedef struct node_struct node_t;
```

- continuing this example, we can define the head or beginning of a linked list as follows:

```
node_t *pHead;
pHead = (node_t *) malloc (sizeof (node_t));
pHead -> pNext = NULL;
```

- malloc(·) is a C built-in function which returns a void pointer to a block of memory which it allocates (as a side-effect) from an area of program memory known as the heap.

- the counterpart of malloc(·) is the function free(·) which returns void and has the side-effect of de-allocating the memory (freeing it to be used by the program for other purposes) pointed to by its argument

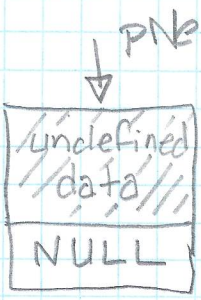
- for example:

```
free((void *) pHead);
```

de-allocates the memory for the single node we created above.

- createNode(·): a function to create new nodes

```
node_t *  createNode(void)
{
        node_t  *pNew = (node_t *) malloc(sizeof(
                                          node_t));
        pNew -> pNext = NULL;
        return pNew;
}
```



# Data Structures : Binary Search Trees



- a binary search tree can operate as a dictionary (ordered data that can be efficiently searched) or a priority queue (allowing important data to be inserted for later efficient retrieval)
- organization:
  - → each node in the tree has one parent and (at most) two children, a left child and a right child.
  - → each node consists of:
    - i) a key (the node's identifier)
    - ii) satellite data (can be anything required by the application, e.g. addresses, sensor data, images, video or sound samples, etc.)
    - iii) pointer to left child
    - iv) pointer to right child
    - v) pointer to parent
  - → the keys determine node placement:

BINARY-SEARCH-TREE PROPERTY:

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$

- eg. (C-language node definition for BST)

```
struct node_struct_simple
{
    unsigned int key;
    struct node_struct_simple *pL;
    struct node_struct_simple *pR;
    struct node_struct_simple *pParent;
};
typedef struct node_struct_simple bst_t;
```

```
bst_t *pHead = NULL;
```

Note that a new node can be created using a createNode algorithm. In C:

```
bst_t *createNode (void)
{
    bst_t * pNode = malloc( sizeof (bst_t));

    pNode -> pL = NULL;
    pNode -> pR = NULL;
    pNode -> pParent = NULL;

    return pNode;
}
```

## Ordered Traversal

- a BST may be searched using recursive algorithms; the most commonly used are:
  i) inorder walk
  ii) pre-order walk
  iii) post-order walk
- the inorder walk visits each node, retrieving data in order of smallest-key-value to largest-key-value.
- pseudo-code:

```
inorder-tree-walk (phead):
    if phead ≠ NULL
        inorder_tree_walk (phead.pL)
        print phead.key
        inorder_tree_walk (phead.pR)
```

- addNode(·) algorithm

```
#include <stdlib.h>
:
int main(void)
{
    key_t  loadarr[] = {12, 7, 6, 1, 23, 3, 2};
    bst_t  *pHead = NULL;
    for (size_t i = 0; i != sizeof(loadarr)/
                sizeof(key_t); ++i )
    {
        addnode(pHead, loadarr[i]);
    }
    :
    return 0;
}
```

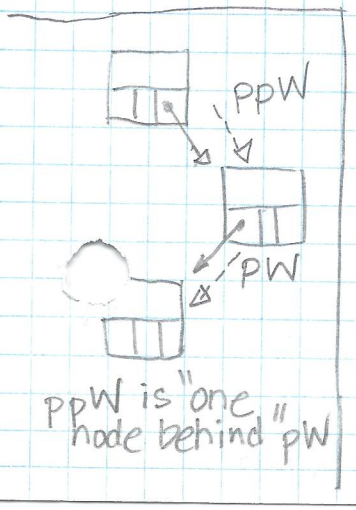
typedef unsigned int key_t;

pseudo-code for addNode(·)
1) create new node
2) attach new node to tree

```
bst_t*  addnode(bst_t* pH, key_t keyval)
{
    bst_t pNew = createNode();
    pNew->key = keyval;
    if (pH == NULL) {
        pH = pNew;
        return pH;
    }
    bool_t right; /* flag indicating which child receives node*/
    bst_t *pW = pH; /* working pointer*/
    bst_t *ppW = pH; /* "parent" of pW */
    while (pW != NULL)
    {
        ppW = pW;
        if (pW->key > keyval)
        {
            pW = pW->pR;
            right = 1;
        }
        else /* pW->key < keyval */
        {
            pW = pW->pL;
            right = 0;
        }
    }
    /* pW is NULL and ppW is either NULL or
       one step behind pW */
```

typedef unsigned short int bool_t;

we assume unique key values in the BST!!



addNode(·) CONT'D

```
    /* add new node to correct child */
    if (right)
    {
        ppW -> pR = pNew;
    }
    else
    {
        ppW -> pL = pNew;
    }
    pNew -> pParent = ppW; /* set parent pointer
                              of new node */
    return pH;
}
```