**Selection / linear search**

Each item in the list is examined in the order it occurs in the list until the desired item is found or the end of the list is reached.

```c
struct email **list;
struct email *start;
*list = start;
list = &start;
```

**Sequential / linear search (code)**

preprocessor directives
function prototypes
int main(){
    struct email *start, *temp;
    int messNo;
    createList(&start);
    printf("Enter the message No. for the search.\n");
    scanf("%d", &messNo);
    temp = searchList(start, messNo);
    if (temp) printf("The list is empty or the message is not in the list.\n");
    else printf("The title of the message is %s.\n", temp->title);
    return 0; }
```
Searching a list -- searchList()

```c
struct email *searchList (struct email *p, int mno){
    struct email *temp;
    temp = p;
    while (temp != NULL && (temp->message_no != n)) {
        temp = temp->fwd;
    }
    return temp;
}

or

struct email *searchList (struct email *p, int mno){
    struct email *temp;
    for(temp=p; (temp != NULL)&&(temp->message_no != n);
        temp =temp->fwd){
    }
    return temp;
}
```

Time complexity analysis

Active operation: Comparison of target message No and the message No in a node.
There are n nodes in a list.
Time complexity function f(n).
The best case scenario: search for message No.1. Or the list is empty.

\[ f(n)_{\text{best}} = 1 \]

The worst case scenario: search for message No.12 or message No.7.

\[ f(n)_{\text{worst}} = n \]

Average \( f(n)_{\text{avg}} = n/2 \)
**Binary search search**

Starting with an ordered array, the desired item is first compared to the middle element in the list (for lists with even number of elements, either of the two middle elements can be used).

---

**Binary search an array of emails (code)**

```c
preprocessor directives
function prototypes
int main(){
    struct email *array;
    int messNo, temp;
    createArray(array);
    printf("Enter the message No. for the search.\n");
    scanf("%d", &messNo);
    temp = searchArray(start, messNo);
    if (temp == -1) printf("the message is not in the array.\n");
    else printf("The title of the message is \n%s\", array[temp].title);
    return 0; }
```

Struct email {
    int message_no;
    char title[40];
}
**Searching an array -- searchArray()**

```c
/* Pre: The contiguous array has been created
Post: If an entry in the array has key equal to the
target, then the function returns the index of
the entry (success). Otherwise the function
returns -1 (target is not found). */
int searchArray (struct email *p, int mno, int size){
    int bottom, top, middle=-1;
    bottom = 0;
    top = size-1;

    while (bottom <= top){
        middle = (top + bottom)/2;
        if (mno == p[middle].messageNo) return index;
        else if (mno < p[middle].messageNo) top = middle-1;
        else bottom = middle +1;
    }
    return -1;
}
```

**Time complexity analysis**

Active operation: Comparison of target message No and
the message No in a node.
There are n nodes in the array.
Time complexity function f(n).
The best case scenario: search for message No.5 (for the
implementation). f(n)_best = 1
The worst case scenario: search for message No.12 or
message No.7.

\[
f(n)_{\text{worst}} = \log_2 n
\]

Average f(n)_{avg} = \log_2 n - 1
Comparison of binary and sequential search

More examples

Free trees

rooted trees
Decisions

- Up to 2 children and
  interior nodes containing data
  ---- binary trees

- Up to M children and
  interior nodes containing only search info
  ---- B-trees

- Other decisions ---- other types of trees

Binary trees

Maximum number of children is 2.
Interior nodes contain actual data.

Definition: A binary tree is either empty, or it consists of
a node called the root together with two binary trees
called the left subtree and the right subtree of the
root.

A node having no a parent node ---- rootNode
A node having no a child node ---- leafNode
A node having both a parent node and a child node ----
interior node
**Binary trees (cont.)**

```c
Struct BinNode *treeStart;
```

Empty tree:
```
    treeStart = NULL
```

Tree having a single node:
```
    treeStart -> leftchild = treeStart -> rightchild = NULL
```

Normal binary tree:
```
        treeStart
          /    \\
         /     \\
       /       \\
      /         \\
     /           \\
```

---

**Traversal of binary trees**

Traversal – moving through all the nodes of a binary tree, visiting each one in turn.

The action we shall take when we visit each node will depend on the application.

Standard traversal orders:
- VLR (preorder)
- LVR (inorder)
- LRV (postorder)

Where V,L, and R indicate the task of visiting a node, traversing the left subtree and traversal the right subtree, respectively.
**Traversal of binary trees (code)**

Void travInorder (BinNode *root, void (*Visit)(struct trec x))
{
    if (root)
    {
        travInorder(root->leftchild, Visit);
        Visit(root -> entry);
        travInorder(root -> rightchild, Visit);
    }
}

Note: a function Visit handling the action in a visit to a node, we can declare a pointer Visit pointing to the function as:
void (*Visit)(struct trec x);

**Recursion**

Recursion is an important programming mechanism. A definition of a function or other objects refers to the function or the object itself. ---- recursive definition

A call to a recursive function leads to a recursive function call, in which the called function calls itself.

There are two cases in a recursive definition:
Base rule describes the term to be defined directly, and
Recursive rule contains the term to be defined in its own definition.

The base rule is for the termination or stopping. The recursive rule is converting the big problem to a similar but simpler problem.
Examples of recursive definitions

```c
Int fac(int n) {
    if (n==0) return 1
    else return (fac(n-1)*n)
}

Int sum(int n) {
    if (n==0) return 0;
    else return (sum(n-1)+n);
}
```

fac(5) = fac(4)*5  
  = fac(3)*4*5  
  = fac(2)*3*4*5  
  = fac(1)*2*3*4*5  
  = fac(0)*1*2*3*4*5  
  = 1*1*2*3*4*5

Can you try sum(5)?

More example of recursive function call

```c
Void travInorder (BinNode *root, void (*Vist)(struct trec x));
```

cdecl> explain void (*vist)(struct trec);

declare visit as pointer to function (struct trec) returning void

cdecl> declare visit as pointer to function(x as struct trec) returning void;

void (*visit)(struct trec x)

A note about this function is in the notice board for this lecture.
A binary search tree is a binary tree that is either empty or in which every node contains a key and satisfies the following conditions:

1. The key in the left child of a node (if exists) is less than the key in its parent node.
2. The key in the right child of a node (if exists) is greater than the key in its parent node.
3. The left and right subtrees of the root are again binary search trees.

An EXAMPLE of a binary search tree
**SEARCH EXAMPLES**

- Consider the tree below, search for keys 21, 40

```
  30
 /  \
20   50
|     |
10    23
|     |
15    21
```

**Time complexity analysis for searching a B-tree**

Active operation: Comparison of target key value and the key value in a node.

There are $n$ nodes in the B-tree with $M$ being the maximum number of children for each node.

Time complexity function $f(n)$.

For $k$ entries per node, the average hits $f(n)_{\text{avg}} = \log_k(n)$. 