Algorithms and Data Structures 2

3. Searching & File Handling

1. Searching

Searching

- *Searching* is the process of finding a *target element* among a group of items (the *search pool*), or determining that it isn't there.

- This requires repetitively comparing the target to candidates in the search pool.

- An efficient sort performs no more comparisons than it has to.

- The size of the search pool is a factor.

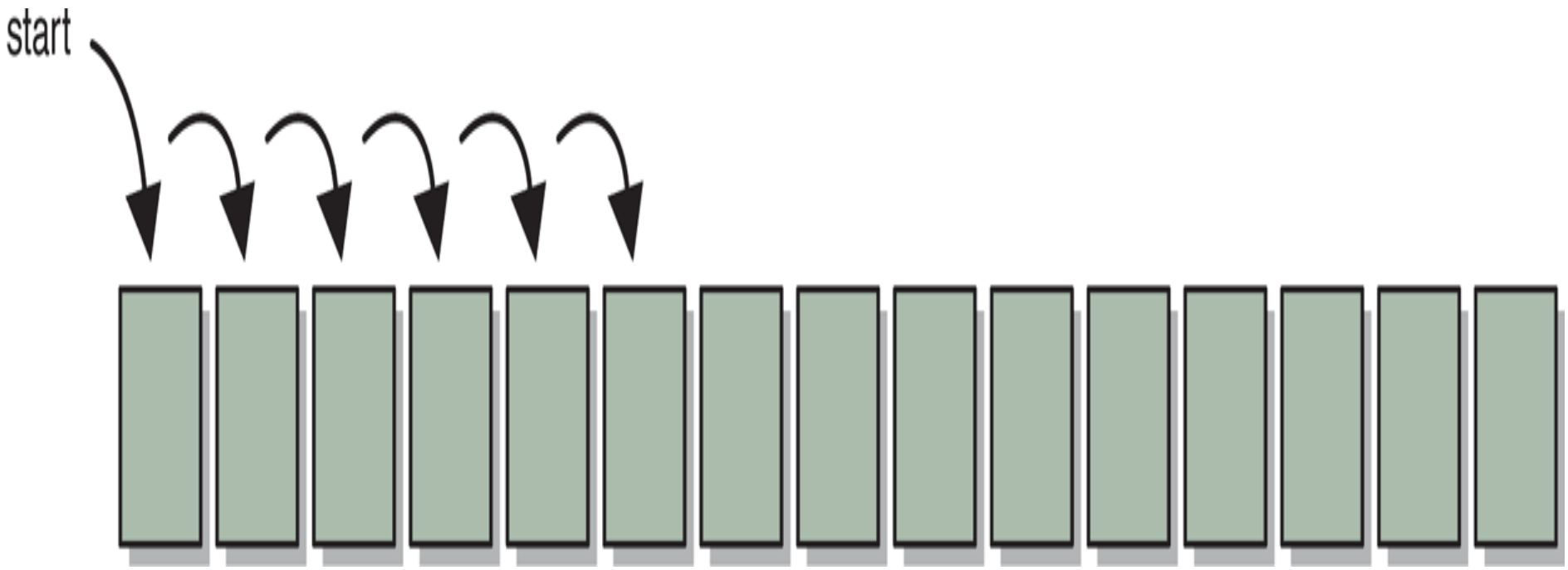
Linear Search

- A *linear search* simply examines each item in the search pool, one at a time, until either the target is found or until the pool is exhausted.

- This approach does not assume the items in the search pool are in any particular order.

- We just need to be able to examine each element in turn (in a linear fashion).
-
- It's fairly easy to understand, but not very efficient

A linear search



Linear Search with an Array

- Assuming the search is being carried out on an array of integers
- Note that in C, it is much more difficult to write truly generic algorithms.
- This is a linear search on an array

```
void linearSearch (int *data,int min,int max, int target)
  {
     int index = min;
     bool found = false;

     while (!found && index <= max)
     {
        if (data[index] == target)
           found = true;
        index++;
     }
```

```
if ( found )
                printf("%d has been found\n", target);
        else
           printf("%d was not found\n", target);
  } //end linearSearch
```

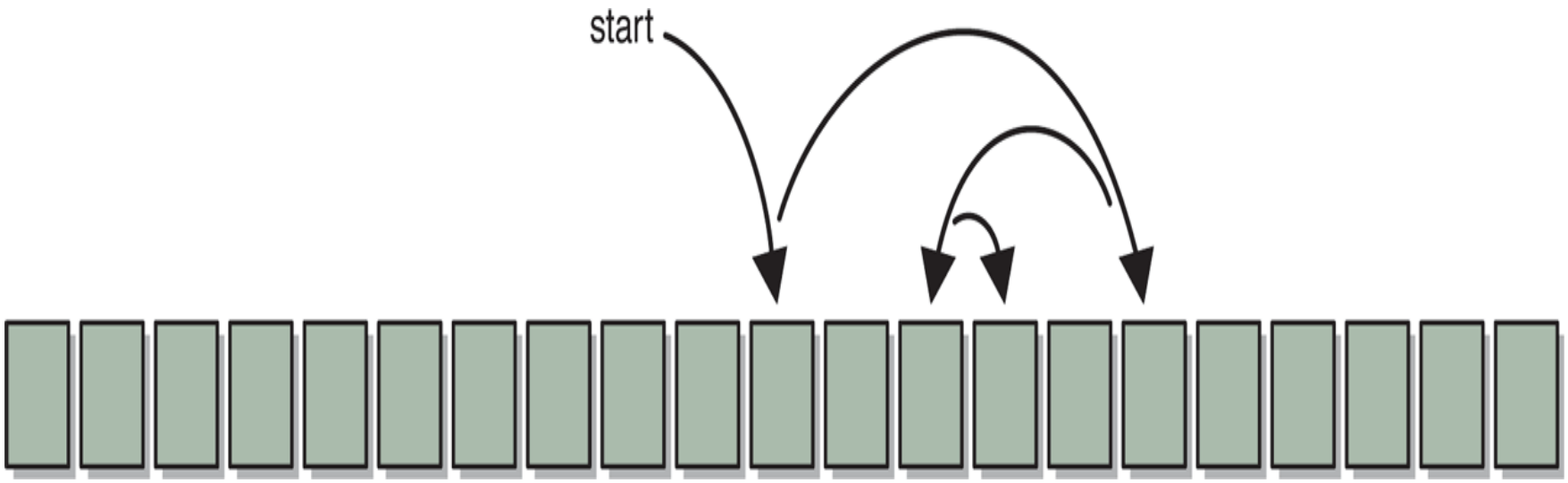
Linear Search with a Linked List

```
void linearSearch(int target)
{
        struct LinearNode *current = list;
        while (current != NULL && current->element->num !=
target)
                                current=current->next;
         if (current == NULL)
            printf("%d does not exist in the list\n", target);
        else
            printf("%d has been found in the list\n", current-
>element->num);
}
```

Binary Search

- If the search pool is sorted, then we can be more efficient than a linear search.
- A *binary search* eliminates large parts of the search pool with each comparison.
- Instead of starting the search at one end, we begin in the middle.
- If the target isn't found, we know that if it is in the pool at all, it is in one half or the other.
- We can then jump to the middle of that half, and continue similarly

A binary search



Note that elements must be SORTED before carrying out the search.

Binary Search

- For example, find the number 29 in the following sorted list of numbers:

  8 15 22 29 36 54 55 61 70 73 88

- Compare the target to the middle value 54

- We now know that if 29 is in the list, it is in the front half of the list

- With one comparison, we've eliminated half of the data

- Then compare to 22, eliminating another quarter of the data, etc.

Binary Search

- A binary search algorithm is often implemented recursively
- Each recursive call searches a smaller portion of the search pool
- The base case of the recursion is running out of *viable candidates* to search, which means the target is not in the search pool
- At any point there may be two "middle" values, in which case either could be used

binarySearch

```
bool binarySearch (int* data, int min, int max, int target)
  {
        bool found = false;
    int midpoint = (min + max) / 2;  // determine the midpoint

    if (data[midpoint] == target)
        found = true;
    else if (data[midpoint] > target)
        {
        if (min <= midpoint - 1)
           found = binarySearch(data, min, midpoint - 1, target);
        } //end else
```

```
else {
    if (midpoint + 1 <= max)
        found = binarySearch(data, midpoint + 1, max, target);
  } //end else
return found;

   } //end binarySearch
```

Comparing Search Algorithms

- On average, a linear search would examine n/2 elements before finding the target
- Therefore, a linear search is O(n)
- The worst case for a binary search is ($\log_2 n$) comparisons

e.g. if n = 8, search would require 3 comparisons
if n = 25, search would require 5 comparisons

- A binary search is a *logarithmic algorithm*
- It has a time complexity of O($\log_2 n$)
- But keep in mind that the search pool must be sorted
- For large n, a binary search is much faster

2. File Handling

File Processing in C

- Like most programming languages, C provides a method for making data storable and permanent. It does this by allowing us to create a text file that stores data between runs of a program.

- There are two types of file in C
  - ASCII (text file) : values are stored as characters and each character uses one byte of storage. Advantage of a text file is that its contents can be viewed by simply opening the file independent of a program. **A text file is used to store lines of text.**
  - binary file: values are stored in binary form. This file takes up less space that an ASCII file but its contents can only be read by the programming language that created it. **A binary file is used to store objects and structures.**

- Files are accessed in C though the use of ***file pointers.***

Text input/output in all languages implicitly use data files

```
printf ("Hello World");
```

is equivalent to

```
fprintf(stdout, "Hello World");
```

```
scanf("%d", &num);
```

is equivalent to

```
fscanf(stdin, "%d", &num);
```

File Syntax

- Declare a file pointer

  ```
  FILE *filepointer;
  ```

- Open a file (*fopen*)
  - This associates a file pointer with a physical file. File must be opened in a specific mode:
    - "r", "w", "a", "r+", "w+", "a+"

    e.g. `filepointer = fopen("file1.txt", "r+");`

  - If file can be opened (i.e. it exists or can be created), then `filepointer` holds the address at the start of the file otherwise it has value NULL.

–As data is being added to the file, filepointer is updated to point to the next available storage area in the file.

–To open a file in **binary mode (for storing structures)**, just add *b* to mode.

e.g. filepointer = fopen("file1.txt", "r+b");

filepointer = fopen("file1.txt", "rb");

filepointer = fopen("file1.txt", "w+b");

filepointer = fopen("file1.txt", "wb");

•Every opened file must be closed before the end of the program.

–e.g. fclose(filepointer);

Block Input and Output

- This allows a block of memory (a record/object) to be read/written to a file (binary file).

```
num_read =
        fread (mem_ptr, size, num, file_ptr)

num_written =
        fwrite(mem_ptr, size, num, file_ptr)
```

Example - storing students records in a file

```
#include <stdio.h>
#define MAX 20

// Structure Template
struct student {
    char studentId[10];
    char name[40];
    char course[5];
    int year;
};
```

```
//Function prototypes
void inputAStudent(int);
void saveToFile(FILE *);
void getFromFile(FILE *);
void displayStudents();

// Global variables
struct student classOfStudents[MAX];
```

```
void main()
{
        FILE *fp;
        int i;

        if ((fp = fopen("h:students.dat", "rb")) == NULL) {
        //get user to input student records
                        for (i=0; i< MAX; i++)
                                inputAStudent(i);
        }
        else
                getFromFile(fp);

        displayStudents();

        saveToFile(fp);
}
```

```
void inputAStudent(int i) {
        printf("Enter student id : ");
        scanf("%s",  classOfStudents[i].studentId);
        etc.
}
void saveToFile(FILE *fp) {
        fp = fopen("h:students.dat", "wb");
        int i;
        for (i=0; i< MAX; i++)
fwrite(&classOfStudents[i], sizeof(struct student), 1, fp);
        fclose(fp);
}
```

```
void getFromFile(FILE *fp)  {
        int i = 0;

        printf("Retriving students from file...\n");
        while (fread(&classOfStudents[i], sizeof(struct
                                                student), 1, fp) != 0)
                        i++;
        fclose(fp);
}

void displayStudents() {
        for (int i=0; i< MAX; i++)
                printf("Student %d is %s\n", i+1,
                        classOfStudents[i].studentId);
        etc.
}
```

Rewinding a File

- The rewind() function resets the file pointer back to the start of the file.

```
rewind(filepointer);
```

- In the previous example, rather than closing and reopening the file in the middle of the program, the rewind command could have been used.

```
if ((fp = (fopen("h:student.dat", "r+b")) == NULL)
                                etc.
        rewind(fp);
```

- Then the file does not need to be reopened for writing at the start of saveToFile.

Random Access of all Files

- The fread/fwrite commands will only allow you to perform serial file processing.

- To allow random/direct access, need to be able to move the file pointer to any position in the file before reading/writing.

```
fseek(filepointer, offset, origin)
```

  - where offset is of type long int and it instructs the filepointer of its new position.

–origin determines the point from where the offset is measured.

- SEEK_SET - measure from beginning
- SEEK_CUR - measure from current position
- SEEK_END - measure from end of file.

- `fseek` returns 0 if it executes successfully, it returns a non-zero integer otherwise.

Example - Assessing a file containing supplier details

CODE	NAME	BALANCE
1	Fruit Suppliers Ltd.	110.00
2	Southern Dairies Ltd.	210.50
3	P.D. Baker	155.50
.....		
20	P. Last	45.91

- The data is held in the supplier file **supplier.dat**, with each supplier code corresponding to the position of the supplier record in the file.

- The following program assumes that the records have already been input into the **supplier.dat** file.

```
#include <stdio.h>
#define MAX_RECS 20

//function prototypes
long getInputCode(long, FILE *);

void displaySupplier(long, FILE *);

FILE *openFile(char  *, char *);
```

```
// Structure Template
struct supplierRecord
{
  long code;
  char name[31];
  float balance;
};


//Global variables
File *fp;
```

```
void main () {
        long inCode;
        fp = openFile("supplier.dat", "rb");

        do
        {
          inCode = getInputCode();
          if (inCode > 0 && inCode <= MAX_RECS)
            displaySupplier(inCode, fp);
        }
        while (inCode > 0);
        fclose(fp);
}
```

```
void displaySupplier(long suppCode, FILE *fp)
{
struct supplierRecord supp;
long offset;

offset = (suppCode - 1) * sizeof(struct supplierRecord);
fseek(fp, offset, SEEK_SET);
if ((fread(&supp, sizeof(struct supplierRecord), 1, fp))
                                                != 1)
  printf("\nError in reading file");
else
  printf("\nCode: %d\nName: %s\nBalance: %7.2f\n",
         supp.code, supp.name, supp.balance);
}
```

```
long getInputCode()
{
long suppCode;

printf("Enter a supplier code from 1 to %d", MAX_RECS);
scanf("%ld", &suppCode);
return(suppCode);
}
```

```
FILE *openFile(char *file, char *mode)
{
    FILE *fp;
    if ((fp = fopen(file, mode)) == NULL)
        printf("Cannot open file %s", file);
    return(fp);
}
```