

Data Structures

Introduction

Prerequisites

- Students must have passed BIL105E Introduction to Scientific and Engineering Computing (C).
- **Knowledge of the C language** and having written programs (even if they were small examples) is an absolute must.
- Students who do not have knowledge of C have zero chance of being successful in Data Structures C.

Data Structures

- In computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.
- A data structure is a mathematical and logical organization of data.
- For an algorithm to work efficiently, it has to use a well-designed data structure.
- A well-designed data structure allows a variety of critical operations to be performed using as little resources (e.g., execution time and memory space) as possible.

Data Structures

- In the design of large systems, it has been observed that the quality and performance of the final result depends heavily on choosing the best data structure.
- After the data structures are chosen, the designing of the algorithm becomes a relatively easier task.

Data Structures

- Array
- List
- Stack
- Queue
- Tree
- ...

Introduction of the Lecture Example

PHONE BOOK

We will realize a phone book application that displays the numbers of recorded people, adds records, deletes records, updates records, and searches for records.

Lecture codes can be found on [Ninova](#).

Lecture Examples

- In the sample code, the C programming language and structured programming will be used.
- For compatibility with new standards, we will make use of some novelties that the C++ language brings.
- These properties will be indicated where they are used.

Example:

C

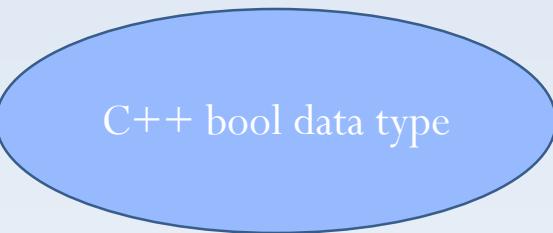
```
printf("%d\n", number);
scanf("%d", &number);
```

C++

```
cout << number << endl;
cin >> number;
```

Phone Book

```
int main() {  
    bool end = false;  
    char choice;  
    while (!end) {  
        print_menu();  
        cin >> choice;  
        end = perform_operation(choice);  
    }  
    return EXIT_SUCCESS;  
}
```



menu_print

```
void menu_print(){
    system("clear");
    cout << endl << endl;
    cout << "Phone Book Application" << endl;
    cout << "Choose an operation" << endl;
    cout << "S: Record Search" << endl;
    cout << "A: Record Add" << endl;
    cout << "U: Record Update" << endl;
    cout << "D: Record Delete" << endl;
    cout << "E: Exit" << endl;
    cout << endl;
    cout << "Enter a choice {S,A,U,D,E}: ";
}
```

menu_print

```
ca C:\Documents and Settings\Sanem Kabadayı\My Do
Phone Book Application
Choose an operation
S: Record Search
A: Record Add
U: Record Update
D: Record Delete
E: Exit

Enter a choice {S, A, U, D, E} : _
```

perform_operation

```
bool perform_operation(char choice){  
    bool terminate=false;  
    switch (choice) {  
        case 'S': case 's':  
            search_record();  
            break;  
        case 'A': case 'a':  
            add_record();  
            break;  
        case 'U': case 'u':  
            update_record();  
            break;  
        case 'D': case 'd':  
            delete_record();  
            break;  
        case 'E': case 'e':  
            cout << "Are you sure you want to exit the program? (Y/N):";  
            cin >> choice;  
            if(choice=='Y' || choice=='y')  
                terminate=true;  
            break;  
        default:  
            cout << "Error: You have entered an invalid choice" << endl;  
            cout << "Please try again {S, A, U, D, E} :" ;  
            cin >> choice;  
            terminate = perform_operation(choice);  
            break;  
    }  
    return terminate;  
}
```

perform_operation

```
bool perform_operation(char choice){
```

```
    switch (choice) {  
        case 'S': case 's':  
            search_record();  
            break;  
        case 'A': case 'a':  
            add_record ();  
            break;  
        case 'U': case 'u':  
            update_record();  
            break;  
        case 'D': case 'd':  
            delete_record();  
            break;  
    }  
}
```

perform_operation

```
case 'E': case 'e':  
    cout << "Are you sure you want to exit the  
    program? (Y/N):";  
    cin >> choice;  
    if(choice=='Y' || choice=='y')  
        terminate=true;  
    break;  
default:  
    cout << "Error: You have entered an invalid choice"  
        << endl;  
    cout << "Please try again {S, A, U, D, E} :";  
    cin >> choice;  
    terminate = perform_operation(choice);  
    break;  
}  
return terminate;
```

File Structure

- Data stored in structures in main memory are temporary and they disappear with the ending of the program.
- Files provide an environment where we can persistently store data.
- Computers store files in secondary storage units (hard disks, magnetic disks, optical disks, ...).
- Performing operations in secondary storage units is slower than in main memory.

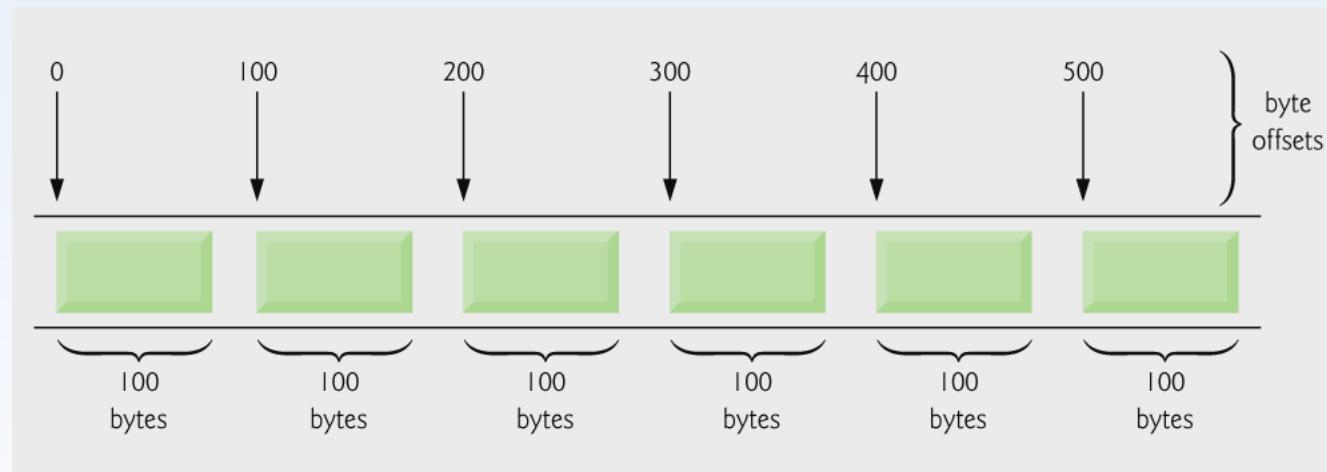
Record Structure

```
#define NAME_LENGTH 30
#define PHONENUM_LENGTH 15

struct Phone_Record{
    char name[NAME_LENGTH];
    char phonenum[PHONENUM_LENGTH];
};
```

Data Structure – 1. week File

- Random access files (reminder)
 - Provide direct access to records
 - Work on fixed-length records



C++ and Struct

- The struct structure of the C++ language, provides a natural capsule in defining abstract data types.
- Thus, the data type and the functions that define the operations that can be performed on this type are located in the same capsule and are logically linked.

C

```
typedef struct DataType{  
    int array[10];  
    int elementnumber;  
} NewArrayType;  
  
int ElementCount(NewArrayType  
*d) {  
    return d->elementnumber;  
}
```

C++

```
struct DataType{  
    int array[10];  
    int elementnumber;  
    int ElementCount();  
};  
  
int DataType::ElementCount() {  
    return elementnumber;  
}
```

Data Structure– 1. week File

“fileoperations.h”

```
struct File{  
    char *filename;  
    FILE *phonebook;  
    void create();  
    void close();  
    void add(Phone_Record *);  
    int search(char []);  
    void remove(int recordnum);  
    void update(int recordnum, Phone_Record *);  
};
```

Data Structure

```
typedef File Datastructure;
Datastructure book;

int main(){
    book.create();
    bool end = false;
    char choice;
    while (!end) {
        print_menu();
        cin >> choice;
        end = perform_operation(choice);
    }
    book.close();
    return EXIT_SUCCESS;
}
```

Data Structure– 1. week File

“fileoperations.h”

```
struct File {  
    char *filename;  
    FILE *phonebook;  
    void create();  
    void close();  
    void add(Phone_Record *);  
    int search(char []);  
    void remove(int recordnum);  
    void update(int recordnum, Phone_Record *);  
};
```

Data Structure “create”

```
void File::create(){
    filename="phonebook.txt";
    phonebook = fopen( filename, "r+" );
    if(!phonebook){
        if(! (phonebook=fopen(filename,"w+"))){
            cerr << "Cannot open file" << endl;
            exit(1);
        }
    }
}
```

Data Structure– 1. week File

“fileoperations.h”

```
struct File {  
    char *filename;  
    FILE *phonebook;  
    void create();  
    void close();  
    void add(Phone_Record *);  
    int search(char []);  
    void remove(int recordnum);  
    void update(int recordnum, Phone_Record *);  
};
```

Data Structure “close”

```
void File::close (){  
    fclose(phonebook);  
}
```

perform_operation

```
bool perform_operation(char choice){  
    bool terminate=false;  
    switch (choice) {  
        case 'S': case 's':  
            search_record();  
            break;  
        case 'A': case 'a':  
            add_record();  
            break;  
        case 'U': case 'u':  
            update_record();  
            break;  
        case 'D': case 'd':  
            delete_record();  
            break;  
        case 'E': case 'e':  
            cout << "Are you sure you want to exit the program? (Y/N):";  
            cin >> choice;  
            if(choice=='Y' || choice=='y')  
                terminate=true;  
            break;  
        default:  
            cout << "Error: You have entered an invalid choice" << endl;  
            cout << "Please try again {S, A, U, D, E} :";  
            cin >> choice;  
            terminate = perform_operation(choice);  
            break;  
    }  
    return terminate;  
}
```

add_record

```
void add_record(){  
    Phone_Record newrecord;  
    cout << "Please enter contact information  
          you want to add" << endl;  
    cout << "Name : " ;  
    cin.ignore(1000, '\n');  
    cin.getline(newrecord.name,NAME_LENGTH);  
    cout << "Phone number :";  
    cin >> setw(PHONENUM_LENGTH)  
        >>newrecord.phonenum;  
    book.add(&newrecord);  
    cout << "Record added" << endl;  
    getchar();  
};
```

C:\Documents and Settings\Sanem Kabadayı\My Documents\Vis

Phone Book Application

Choose an operation

S: Record Search

A: Record Add

U: Record Update

D: Record Delete

E: Exit

Enter a choice <S, A, U, D, E> : a

Please enter contact information you want to add

Name : Ahmet Gü

Phone number :02123455454

Data Structure – 1. week File

“fileoperations.h”

```
struct File{  
    char *filename;  
    FILE *phonebook;  
    void create();  
    void close();  
    void add(Phone_Record *);  
    int search(char []);  
    void remove(int recordnum);  
    void update(int recordnum, Phone_Record *);  
};
```

add

```
void File::add(Phone_Record *nrptr){  
    fseek(phonebook, 0, SEEK_END);  
    fwrite(nrptr, sizeof(Phone_Record), 1,  
    phonebook);  
}
```

perform_operation

```
bool perform_operation(char choice){  
    bool terminate=false;  
    switch (choice) {  
        case 'S': case 's':  
            search_record();  
            break;  
        case 'A': case 'a':  
            add_record();  
            break;  
        case 'U': case 'u':  
            update_record();  
            break;  
        case 'D': case 'd':  
            delete_record();  
            break;  
        case 'E': case 'e':  
            cout << "Are you sure you want to exit the program? (Y/N):";  
            cin >> choice;  
            if(choice=='Y' || choice=='y')  
                terminate=true;  
            break;  
        default:  
            cout << "Error: You have entered an invalid choice" << endl;  
            cout << "Please try again {S, A, U, D, E} :";  
            cin >> choice;  
            terminate = perform_operation(choice);  
            break;  
    }  
    return terminate;  
}
```

search_record

```
void search_record(){
    char name[NAME_LENGTH];
    cout << "Please enter the name of the person
you want to search for (press '*' for full
list):" << endl;
    cin.ignore(1000, '\n');
    cin.getline(name, NAME_LENGTH);
    if(book.search(name)==0){
        cout << "Could not find a record
matching your search criteria" << endl;
    }
    getchar();
};
```

C:\Documents and Settings\Sanem Kabadayı\My Documents\Visual Studio 2008\Projects\ph... - □

Phone Book Application

Choose an operation

S: Record Search

A: Record Add

U: Record Update

D: Record Delete

E: Exit

Enter a choice <S, A, U, D, E> : s

Please enter the name of the person you want to search for (press '*' for full list):

*

- 1. Ahmet Gül 02123455454
- 2. Mehmet Şahin 02123121212
- 3. Ali Veli 05324411212

C:\Documents and Settings\Sanem Kabadayı\My Documents\Visual Studio 2008\Projects\ph... [-] [x]

Phone Book Application

Choose an operation

S: Record Search

A: Record Add

U: Record Update

D: Record Delete

E: Exit

Enter a choice (S, A, U, D, E) : s

Please enter the name of the person you want to search for (press '*' for full list):

a

1.Ahmet Gül 02123455454

3.Ali Veli 05324411212

-

Data Structure – 1. week File

“fileoperations.h”

```
struct File{  
    char *filename;  
    FILE *phonebook;  
    void create();  
    void close();  
    void add(Phone_Record *);  
    int search(char []);  
    void remove(int recordnum);  
    void update(int recordnum, Phone_Record *);  
};
```

```
int File::search(char *desired){  
    Phone_Record k;  
    int index=0;  
    bool all=false;  
    int found=0;  
    if(strcmp(desired,"*")==0)  
        all=true;  
    fseek(phonebook, 0, SEEK_SET);  
    while(!feof(phonebook)){  
        index++;  
        fread( &k, sizeof (Phone_Record) , 1, phonebook);  
        if(feof(phonebook)) break;  
        if(all || strnicmp(k.name,desired,strlen(desired))==0){  
            cout << index << ". " << k.name << " " << k.phonenum  
            << endl;  
            found++;  
        }  
    }  
    return found;  
}
```

perform_operation

```
bool perform_operation(char choice){  
    bool terminate=false;  
    switch (choice) {  
        case 'S': case 's':  
            search_record();  
            break;  
        case 'A': case 'a':  
            add_record();  
            break;  
        case 'U': case 'u':  
            update_record();  
            break;  
        case 'D': case 'd':  
            delete_record();  
            break;  
        case 'E': case 'e':  
            cout << "Are you sure you want to exit the program? (Y/N):";  
            cin >> choice;  
            if(choice=='Y' || choice=='y')  
                terminate=true;  
            break;  
        default:  
            cout << "Error: You have entered an invalid choice" << endl;  
            cout << "Please try again {S, A, U, D, E} :";  
            cin >> choice;  
            terminate = perform_operation(choice);  
            break;  
    }  
    return terminate;  
}
```

update_record

```
void update_record(){
    char name[NAME_LENGTH];
    int choice;
    cout << "Please enter the name of the person whose record you want
          to update (press '*' for full list):" << endl;
    cin.ignore(1000, '\n');
    cin.getline(name, NAME_LENGTH);
    int personcount=book.search(name);
    if(personcount==0){
        cout << "Could not find a record matching your search criteria"
        << endl;
    }
    else {
        if (personcount==1){
            cout << "Record found." << endl;
            cout << " If you want to update this record please enter
                  its number (Enter -1 to exit without
                  performing any operations): " ;
        }else cout << "Enter the number of the record you want to
          update (Enter -1 to exit without performing any operations): ";
    }
}
```

```
    cin >> choice;
    if(choice==1) return;
    Phone_Record newrecord;
    cout << "Please enter current contact
              information" << endl;
    cout << "Name : ";
    cin.ignore(1000, '\n');
    cin.getline(newrecord.name,NAME_LENGTH);
    cout << "Phone number :";
    cin >> setw(PHONENUM_LENGTH) >> newrecord.phonenum;
    book.update(choice,&newrecord);
    cout << "Record successfully updated" << endl;
}
getchar();
};
```

C:\Documents and Settings\Sanem Kabadayı\My Documents\Visual Studio 2008\Projects\ph...

```
Phone Book Application
Choose an operation
S: Record Search
A: Record Add
U: Record Update
D: Record Delete
E: Exit
```

```
Enter a choice {S, A, U, D, E} : u
```

```
Please enter the name of the person whose record you want to update (press '*' for full list):
```

```
a
```

```
1.Ahmet Gül 02123455454
```

```
3.Ali Veli 05324411212
```

```
Enter the number of the record you want to update (Enter -1 to exit without performing any operations): 1
```

```
Please enter current contact information
```

```
Name : a_
```

```
void File::update(int recordnum, Phone_Record
*nrptr){
if(fseek(phonebook,
        sizeof(Phone_Record)*(recordnum-1) ,
        SEEK_SET) == 0)
fwrite(nrptr, sizeof(Phone_Record) , 1 ,
       phonebook);
}
```

perform_operation

```
bool perform_operation(char choice){  
    bool terminate=false;  
    switch (choice) {  
        case 'S': case 's':  
            search_record();  
            break;  
        case 'A': case 'a':  
            add_record();  
            break;  
        case 'U': case 'u':  
            update_record();  
            break;  
        case 'D': case 'd':  
            delete_record();  
            break;  
        case 'E': case 'e':  
            cout << "Are you sure you want to exit the program? (Y/N):";  
            cin >> choice;  
            if(choice=='Y' || choice=='y')  
                terminate=true;  
            break;  
        default:  
            cout << "Error: You have entered an invalid choice" << endl;  
            cout << "Please try again {S, A, U, D, E} :";  
            cin >> choice;  
            terminate = perform_operation(choice);  
            break;  
    }  
    return terminate;  
}
```

delete_record

```
void delete_record(){
    char name[NAME_LENGTH];
    int choice;
    cout << "Please enter the name of the person whose
          record you want to delete (press '*' for full
          list):" << endl;
    cin.ignore(1000, '\n');
    cin.getline(name, NAME_LENGTH);
    int personcount=book.search(name);
    if(personcount==0){
        cout << " Could not find a record matching your
              search criteria " << endl;
    }
}
```

```
else {
    if (personcount==1){
        cout << "Record found." << endl;
        cout << "If you want to delete this record
            please enter its number (Enter -1 to exit
            without performing any operations): " ;
    }
    else cout << "Enter the number of the record you want
        to delete (Enter -1 to exit without
        performing any operations): " ;
    cin >> choice;
    if(choice==-1) return;
    book.remove(choice);
    cout << "Record deleted" << endl;
}
getchar();
};
```

```
void File::remove(int recordnum){  
    Phone_Record emptyrecord={"", ""};  
    if(fseek(phonebook,  
            sizeof(Phone_Record)*(recordnum-  
1), SEEK_SET)==0)  
        fwrite(&emptyrecord, sizeof(Phone_Record),  
              1, phonebook);  
}
```

phoneprog.cpp

```
#include <iostream>
#include <stdlib.h>
#include <iomanip>
#include <ctype.h>
#include "fileoperations.h"

using namespace std;

typedef File Datastructure;
Datastructure book;

void print_menu();
bool perform_operation(char);
void search_record();
void add_record();
void delete_record();
void update_record();

int main(){ ...
```

record.h

```
#define NAME_LENGTH 30
#define PHONENUM_LENGTH 15

struct Phone_Record{
    char name[NAME_LENGTH];
    char phonenum[PHONENUM_LENGTH];
};
```

fileoperations.h

```
#ifndef FILEOPERATIONS_H
#define FILEOPERATIONS_H
#include <stdio.h>
#include "record.h"

struct File{
    char *filename;
    FILE *phonebook;
    void create();
    void close();
    void add(Phone_Record *);
    int search(char []);
    void remove(int recordnum);
    void update(int recordnum, Phone_Record *);
};

#endif
```

fileoperations.cpp

```
#include "fileoperations.h"
#include <iostream>
#include <stdlib.h>
#include <string.h>

using namespace std;

void File::add(Phone_Record *nrptr){
    fseek(phonebook, 0, SEEK_END);
    fwrite(nrptr, sizeof(Phone_Record), 1,
    phonebook);
}

void File::create(){ ... }
```

Recitation: To do

- Explanation of compilation steps and compilation operations

Practice Exercise

Realize:

1. The real deletion operation on the lecture example

Data Structures

Pointers and Arrays

Overview

The topics for this week will serve as reminders of what you learned in BIL 105E.

1. Reminder about the pointer construct in the C++ language
2. Reminder about the array construct
3. Study of the relationship between pointers and arrays
4. Function calls
5. Passing pointers and arrays to functions

Pointers

- The pointer variable contains the address information of where another variable is located in memory.
- Normal variables contain a specific value (**direct reference**)
- Pointers contain the address of a variable that has a specific value (**indirect reference**)

Memory

- We may think of a computer's memory as consisting of N bytes with labels 0 through $N - 1$.
 - N represents the total number of bytes of memory that a computer can have.
 - The labels are called **addresses**.

Address

0	1	0	1	0	1	0	1	1
1	1	0	0	0	1	0	1	1
2	1	0	1	0	0	0	1	1
							⋮	
N-3	1	0	1	0	1	0	1	1
N-2	0	0	0	0	1	0	1	1
N-1	1	0	1	0	0	0	1	0

Memory

- The values of variables of a program are stored in one or more **consecutive** bytes of memory.
- For example, the value of the int variable m is usually stored with four bytes.
- In this case, the contents of these four bytes hold a binary representation of m 's value.

Address

$\&m$	1	0	1	0	1	0	0	1
	1	0	0	0	1	0	1	0
	0	0	1	0	1	0	0	1
	1	0	0	1	1	0	1	0

Memory

- The address of the first byte used to store the value of a variable is called the address of the variable.
- In C++, the address of a variable can be obtained using the address operator '&'.
- For example, the address of the `int` variable `m` is `&m`.

Memory

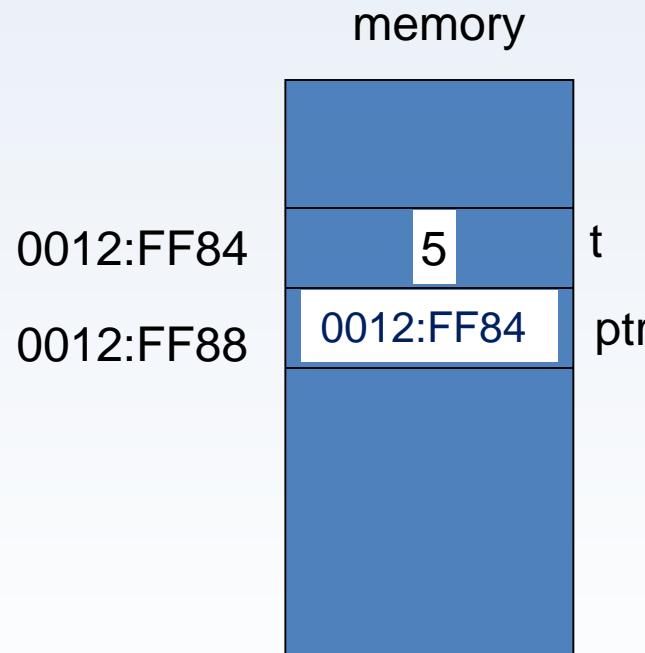
- Memory spaces have addresses that are consecutive.
- These spaces are used as groups of one or more octets. (Lengths of variables may vary from system to system)

32-bit data

char	1 byte
short int	2 bytes
int	4 bytes
float	4 bytes
double	8 bytes

Pointer

- The pointer variable (in 32-bit addressing) takes up 4 bytes in memory space.
- Pointers are all the same size, as they just store a memory address.



```
int t;  
t = 5;  
int *ptr;  
ptr = &t;
```

Pointers

- & sign returns the address of a variable.
- `ptr = &t;` assignment assigns the address of `t` to the `ptr` pointer.
- We say “`ptr` points to `t`.”
-  & sign only returns the variable/array addresses located in memory. It cannot be applied to expressions, constants, or register variables.

ptr: 0012FF84

t: 5

k: 4

&ptr: 0012FF88

&t: 0012FF84

```
• int main(int argc, char* argv[])
  {           int *ptr;
    int t=5;
    int k=sizeof(ptr);
    ptr=&t;
    return 0;
  }
```

Signs

- ***** sign is the indirection operator.
- When ***** is applied to a pointer variable, it accesses the object/data the pointer points to.
- ***ptr = 8;** changes the integer value at the location pointed to by the integer pointer ptr to 8.

ptr: 0012FF88

t: 8

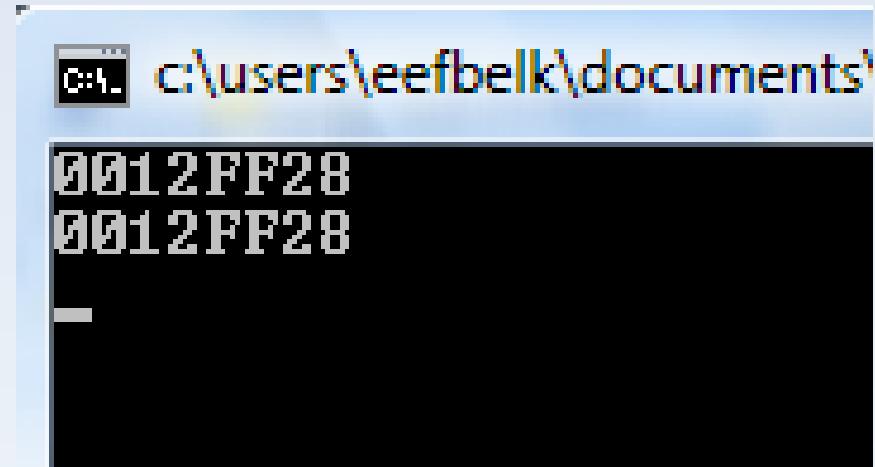
&t: 0012FF88

```
int main(int argc, char* argv[])
{
    int t=5;
    int *ptr=&t;
    *ptr=8;
    return 0;
}
```

* and & are inverses of each other

```
#include <iostream>
```

```
using namespace std;
int main(){
    int t = 5;
    int *ptr = &t;
    cout << &*ptr << endl;
    cout << *&ptr << endl;
    return EXIT_SUCCESS;
}
```



```
0012FF28
0012FF28
```

 t	5
  ptr	0x0012ff28

Pointer operations

- At the end of each operation, the address value gets updated so that it has a variable address of the type it points to.
- For example, if it is a character pointer, the value increment/decrement will be 1 byte; if it is an integer pointer, the value increment/decrement will be 4 bytes.

`ptr++;`

`ptr--;`

- The number of bytes in the variable referenced by `ptr` is added to `ptr`

Pointer operations

- + and - operators can be used on pointers.

```
int *ptr;  
ptr++;
```

008f5838 → 008f583c

- Here, the ++ operation has advanced the pointer by an integer (4 bytes).

Pointer operations

```
char arr[5] = "abcd";
```

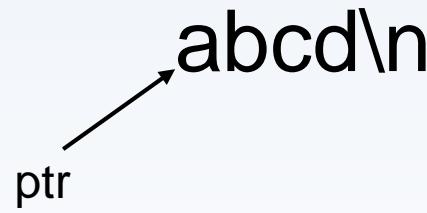
```
char *ptr = arr;
```

```
ptr += 2;
```

```
*ptr = 'x';
```

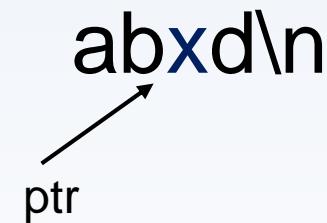
}

```
*(ptr+2) = 'x';
```



abcd\n

ptr



abxd\n

ptr

Question

- `int *ptr;`
- `ptr = ptr + 9;`

By how much is the address `ptr` stores incremented?

4 X number of bytes in the variable referenced by `ptr`

`008f5838` → `008f585c`

Should increase a total of
36 bytes (i.e., 24 in
hexadecimal notation:
 $5838 + 24 = 585c$)

Pointer operations

If `ptr` is pointing to integer `x`, we can use `*ptr` in every context `x` might be used in.

```
int x = 1, y = 2;  
int *ptr;  
ptr = &x;  
y = *ptr;  
*ptr = 0;  
*ptr = *ptr + 10;  
*ptr += 1;  
++*ptr;  
(*ptr)++;
```

Attention to operations

We have to make sure that correct operations are carried out.

- `(*ptr)++;` // increment value in address pointed to by ptr
- `*ptr++ = 5;` // use the value in address pointed to by ptr, then increment ptr
- `*++ptr;` // increment ptr, access value in new address
- `(*++ptr)++;` // increment ptr, access value it points to, and increment that value

Increment and decrement operators

<u>Operator</u>	<u>Called</u>	<u>Sample expression</u>	<u>Explanation</u>
• <code>++</code>	preincrement	<code>++a</code>	Increment a by 1, then use the new value of a in the expression in which a resides.
• <code>++</code>	postincrement	<code>a++</code>	Use the current value of a in the expression in which a resides, then increment a by 1.
• <code>--</code>	predecrement	<code>--b</code>	Decrement b by 1, then use the new value of b in the expression in which b resides.
• <code>--</code>	postdecrement	<code>b--</code>	Use the current value of b in the expression in which b resides, then decrement a by 1.

Increment and decrement operators

- We can write the assignment statement

`x = x + 1;`

- More concisely with assignment operators as

`x += 1;`

- With preincrement operators as

`++x;`

- Or with postincrement operators as

`x++;`

Increment and decrement operators

- Note: when incrementing or decrementing a variable in a statement by itself,
 - the preincrement and postincrement forms have the same effect, and
 - the predecrement and postdecrement forms have the same effect
- It is only when a variable appears in the context of a larger expression that preincrementing the variable and postincrementing the variable have different effects (and similarly for predecrementing and postdecrementing)

Operator precedence and associativity

Operators with a higher level of precedence are higher on the list, and they are carried out before those with a lower order of precedence

<u>Operators</u>	<u>Associativity</u>	<u>Type</u>
• ()	left to right	parentheses
• ++ -- (type)	left to right	unary (postfix)
• ++ -- + - ! & *	right to left	unary (prefix)
• * / %	left to right	multiplicative
• + -	left to right	additive
• << >>	left to right	insertion/extraction
• < <= >= >	left to right	relational
• == !=	left to right	equality
• = += -= *= /= %=	right to left	assignment

```

int t[3] = {1,2,3};
int *ptr = t;
cout << t[0] << "\t" << t[1] << "\t" << t[2] << endl;
*ptr = 8;
(*ptr)++;
cout << t[0] << "\t" << t[1] << "\t" << t[2] << endl;
*ptr++ = 5;
cout << t[0] << "\t" << t[1] << "\t" << t[2] << endl;
*ptr = 6;
cout << t[0] << "\t" << t[1] << "\t" << t[2]<< endl;
(*++ptr)++;
cout << t[0] << "\t" << t[1] << "\t" << t[2] << endl;

```

start	1	2	3
(*ptr)++;	9	2	3
*ptr++=5;	5	2	3
*ptr=6;	5	6	3
(*++ptr)++;	5	6	4

Assigning pointers to each other

- `int *ptr;`
- `int *ip;`
- `ip = ptr;`
- `ip` points to the address `ptr` points to.

Type of variable pointed to

- We must make sure that the pointer variables point to the right type of data.

```
• int main(int argc, char* argv[]) {  
    float x, y;  
    int *p;  
    x = 10.25, y = 20.89;  
    p = &x; // can assign any address to p  
    y = *p;  
  
    return 0;  
}
```

```
x: 10.25  
y: 1.092878E+09  
*p: 1092878336  
p: 0012FF88
```

[C++ Warning] trial.cpp(12): W8075 Suspicious pointer conversion

- At the end of the operation, the `x` value will not be assigned to `y`. This is because `p` has been declared as an integer pointer.
- The operation tries to assign a float value to an integer value and cannot obtain the desired result.

Array Structure

- Arrays are structures that hold related data (same type of data).
- They are static. They remain the same size throughout the program.
- They are made up of successive memory spaces.

- `int a[] = {10, 11, 12, 13};`

Watch 1	
Name	Value
 a	0x0012ff54
 [0]	10
 [1]	11
 [2]	12
 [3]	13
 &a[0]	0x0012ff54
 &a[1]	0x0012ff58
 &a[2]	0x0012ff5c

Two-dimensional arrays

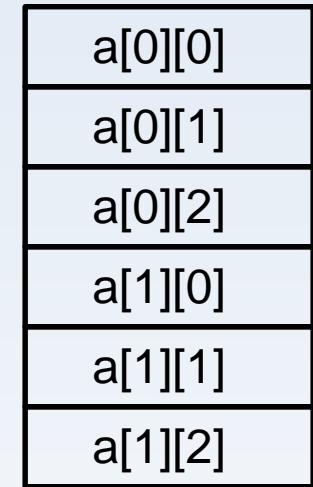
- int a[][][3] = { {1, 2, 3}, {4, 5, 6} };

Watch 1	
Name	Value
a	0x0012ff4c
[0]	0x0012ff4c
[0]	1
[1]	2
[2]	3
[1]	0x0012ff58
[0]	4
[1]	5
[2]	6

	col = 0	col = 1	col = 2
row = 0	a[0][0]	a[0][1]	a[0][2]
row = 1	a[1][0]	a[1][1]	a[1][2]

Two-dimensional arrays

- `int a[][][3] = { {1, 2, 3}, {4, 5, 6} };`
- The two-dimensional array `a` is stored in memory by rows as illustrated below because the operator `'[]'` associates from left to right.
- The position of the component `a[i][j]` in the one-dimensional array is given by $3i + j$, where the positions are labeled from 0 to 6.
- In general, for p rows and q columns, this would be $qi+j$. So, can be accessed using
$$*(\&a[0][0] + q*i + j)$$



2-by-3 array
in memory

Arrays and pointers

- Arrays and pointers are closely related.
 - Array names are pointer constants.
 - Any operation that can be achieved by array indexing can also be done with pointers.
 - Usually, if an array is going to be accessed in strictly ascending or descending order, pointer arithmetic is faster than array indexing.
 - If an array is going to be accessed randomly, array indexing is better.
 - In terms of the underlying address arithmetic, on most architectures it takes one multiplication and one addition to access a one-dimensional array through a subscript.
 - Pointers require no arithmetic at all—they nearly always hold the address of the object that they refer to.

Note! Array name is a constant pointer.

Cannot be changed: a - array name, pa - pointer

pa = a ✓ (the identifier of an array is

treated as a constant address)

a = pa X

pa++ ✓

a++ X

Arrays and pointers

- `int a[10];`
an integer array of 10 elements
 $a[0] \ a[1] \ \dots \ a[9]$
- $a[i] \rightarrow$ a reference to the i th element of array a
- `int *aPtr;`
 $aPtr = \&a[0];$
The pointer takes on a value so that it points to the first element of the array.

Arrays and pointers

- Once “`aPtr = &a[0];`” has been executed, there are five ways that the `third` element (as an example) of the array `a` can be accessed:
 - `a[3]`
 - `aPtr[3]`
 - `*(aPtr + 3)`
 - `*(a + 3)`
 - `*(&a[0] + 3)`

Two-dimensional array example

In a class of 10 students, 3 exams (2 midterms and 1 final) are given throughout the semester. We want to compute the following information using the recorded exam grades:

- Average for first midterm
- Average for second midterm
- Average for final
- Average of students at the end of term
- Class average at the end of term

Two-dimensional array example

	1. Midterm	2. Midterm	Final	Average
0	50	55	45	
1	86	13	60	
2	55	45	75	
3	45	45	10	
4	70	65	76	
5	12	13	10	
6	43	45	80	
7	12	30	35	
8	76	55	65	
9	90	95	98	

We assume that the exams have equal weight.

```
int grades1[10][3] = {{50,55,45}, {86,13,60}, {55,45,75},  
{45,45,10},{70,65,76},{12,13,10},{43,45,80},{12,30,35},  
{76,55,65},{90,95,98}};
```

Name	Value
grades1	0x0012feec
[0]	0x0012feec
[0]	50
[1]	55
[2]	45
[1]	0x0012fef8
[0]	86
[1]	13
[2]	60
[2]	0x0012ff04
[3]	0x0012ff10
[4]	0x0012ff1c
[5]	0x0012ff28
[6]	0x0012ff34
[7]	0x0012ff40
[8]	0x0012ff4c
[9]	0x0012ff58

```
int grades2[3][10] = {{50,86,55,45,70,12,43,12,76,90},  
                      {55,13,45,45,65,13,45,30,55,95},  
                      {45,60,75,10,76,10,80,35,65,98}};
```

Name	Value
grades2	0x0012fe6c
[0]	0x0012fe6c
[0]	50
[1]	86
[2]	55
[3]	45
[4]	70
[5]	12
[6]	43
[7]	12
[8]	76
[9]	90
[1]	0x0012fe94
[2]	0x0012feb8

- int grades1[10][3]

grades1[2][1] → grade student number 3 got on 2. exam

- int grades2[3][10]

grades2[2][1] → grade student number 2 got on 3. exam

grades1[i][j] \leftrightarrow *(grades1 + i*3 + j)

grades2[i][j] \leftrightarrow *(grades2 + i*10 + j)

```

int main(){
    int grades2[3][10] = {{50,86,55,45,70,12,43,12,76,90},
                          {55,13,45,45,65,13,45,30,55,95},{45,60,75,10,76,10,80,35,65,98}};
    float sum = 0, grandsum = 0;
    for (int i = 0; i < 3; i++){           // for each exam of the three exams
        sum = 0;
        for (int j = 0; j < 10; j++) // sum over 10 students for an exam
            sum += grades2[i][j];
        cout << i + 1 << ". exam average=" << sum/10 << endl;
    }
    for (int i = 0; i < 10; i++){      // for each of the 10 students
        sum = 0;
        for (int j = 0; j < 3; j++) // sum over the 3 exams for a student
            sum += grades2[j][i];
        cout << i + 1 << ". student average=" << sum/3 << endl;
        grandsum += (sum/3);           // add the averages of all 10 students
    }
    cout << "Class average=" << (grandsum/10) << endl;
    getchar();
    return EXIT_SUCCESS;
}

```

Two-dimensional array example

In the case of exams having equal weight, the averaging operation is the same for each step:

Average = (sum of numbers to be averaged)/size

```
float average(int *aPtr, int size){  
    int sum = 0;  
    for (int i = 0; i < size; i++){  
        sum += aPtr[i];  
    }  
    return (float)sum/(float)size;  
}
```

Two-dimensional array example

If we only have the function whose prototype is given below to take an average, what kinds of calls should be made to compute the desired values?

```
float average(int *aPtr, int size);
```

- Average of first midterm
- Average of second midterm
- Average of final
- Average of students at the end of term
- Class average at the end of term

Two-dimensional array example

```
for (int i = 0; i < 3; i++)  
    cout << i + 1 << ". exam average ="  
        << average(grades2[i], 10) << endl;
```

This command can be used to compute the averages of exams.

For that purpose, the array declaration should be made as `int grades2[3][10]`.

If the array declaration had been made as

`int grades1[10][3]`, then with a similar `for` loop each student's average could be computed with the following call:

`average(grades1[i], 3)`

Class average at the end of term

In both cases, the class average could be computed as:

(sum of exam averages) / 3

or

(sum of student averages) / 10

To store the data, one of the two structures must be selected:

`int grades1[10][3]` or `int grades2[3][10]`

In this case, it is not possible to compute class averages and student averages by making calls to this function. This is because this function starts from a specific point (the first element of the array passed as a parameter) and operates on consecutive memory slots. This problem can be solved by making small changes to the function.

```
float average(int *aPtr, int size){  
    int sum = 0;  
    for (int i = 0; i < size; i++){  
        sum += aPtr[i];  
    }  
    return (float)sum/(float)size;  
}  
float new_average  
(int *aPtr, int start, int size, int offset)  
{  
    int sum = 0;  
    for (int i = 0; i < size; i++){  
        sum += *(aPtr + start + i*offset);  
    }  
    return (float)sum/(float)size;  
}
```

- Average of first midterm
- Average of second midterm
- Average of final
- Overall average of class at the end of term

```

for (int i = 0; i < 3; i++){
    cout << i + 1 << ". exam average="
        << new_average(&grades2[0][0], i*10, 10, 1)
        << endl;
    sum += new_average(&grades2[0][0], i*10, 10, 1);
}
cout << "Class Average=" << (sum/3) << endl;
for (int i = 0; i < 10; i++)
    cout << i + 1 << ". student's average=" <<
        new_average(&grades2[0][0], i, 3, 10) << endl;

```

Data Structures

Function Calls in C++
Dynamic Memory
Abstract Data Types

Function Calls and the Stack

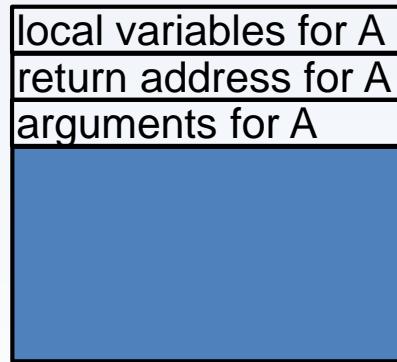
- The stack is an area of memory that is used by a program to communicate with its functions.
- In particular, each time that a function is called, its arguments, its **return address** (the place where the program should return to after the function is finished), and the memory for its local variables are put on top of the stack as shown below.

Each rectangle represents one

or more bytes of memory



the stack before
the call to A



the stack after
the call to A



the stack after the call
to A is completed

The movement of the stack when function A is called

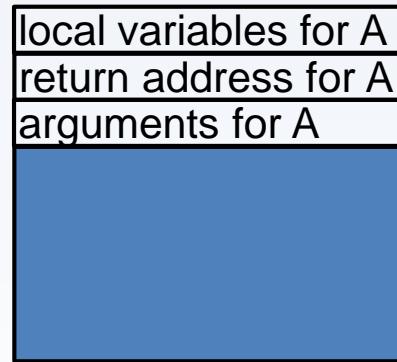
Function Calls and the Stack

- When the function is finished, this information is removed from the stack.
- Notice that when a stack is used in this way, the memory needed for a function's arguments and local variables is **allocated** (that is, put on the stack), only when the function is executing.

Each rectangle represents one or more bytes of memory



the stack before
the call to A



the stack after
the call to A



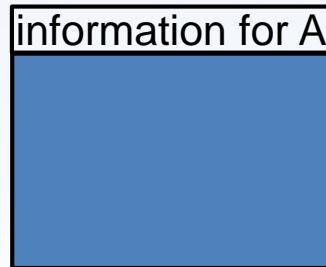
the stack after the call
to A is completed

The movement of the stack when function A is called

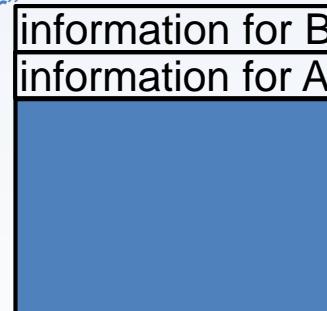
Function Calls and the Stack

- Suppose that function A calls function B, which calls function C, which calls function D.
- This sequence of calls is an example of function **nesting**, and the number of calls in the sequence is called the **nesting depth**.
- The figure below shows how the stack grows when each successive call is made.

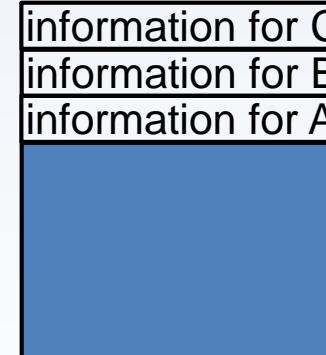
The memory for the arguments, return address and local variables of a function are combined into one rectangle labeled “information for”



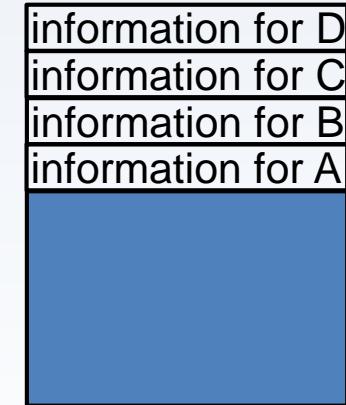
after the call to A



after the call to A
and B



after the call to A,
B, and C

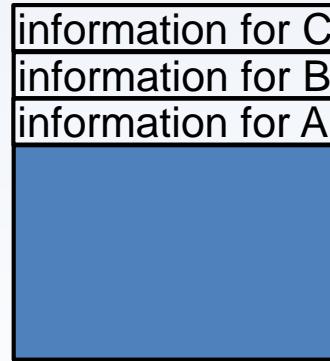


after the call to A,
B, C, and D

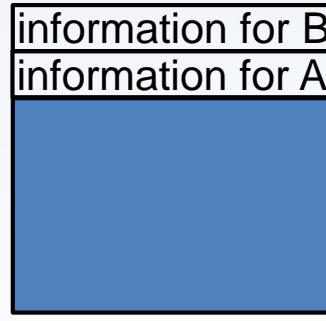
The movement of the stack when functions A, B, C, and D are called

Function Calls and the Stack

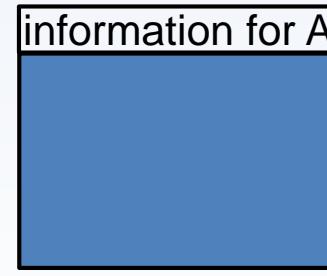
- The figure below shows the movement of the stack when the calls are completed.
- In particular, notice that the functions A, B, C, and D are completed in the **reverse** order in which they are called.



after the call to D
is completed



after the call to C
is completed



after the call to B
is completed



after the call to A
is completed

The movement of the stack after functions A, B, C, and D are completed

Function Calls and the Stack

- We will see how a stack is implemented later.
- For now, we note that the amount of memory allocated to the stack is finite.
- It is possible, particularly with recursion (which we will also see later), to run out stack memory.
- This occurrence called stack overflow, usually causes the program to terminate.

Function Calls in C++

C++ passes arguments to functions by value
(**pass-by-value**).

For that reason, the called function cannot
alter the value of a variable in the calling
program.

Function Calls in C++

- If the function has to make a change to a single value, this value can be passed back to the calling program with a **return** statement.

```
int add(int a, int b){
```

.....

```
    return sum;
```

```
}
```

```
int result = add(5, 7);
```

What if the function has to
change more than one value???

Erroneous program: function only swaps copies of a and b sent to it.

```
void swap(int x, int y){  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}  
  
int main( )  
{    int a = 5, b = 6;  
    swap(a, b);  
    return 0;  
}
```

Correct Function

C STYLE

```
void swap(int *xptr, int *yptr){  
    int temp;  
    temp = *xptr;  
    *xptr = *yptr;  
    *yptr = temp;  
}
```

```
int main( )  
{    int a = 5, b = 6;  
    swap(&a, &b);  
    return 0;  
}
```

C++ STYLE

```
void swap(int &x, int &y){  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}
```

```
int main( )  
{    int a = 5, b = 6;  
    swap(a, b);  
    return 0;  
}
```

Pointers in different programming languages

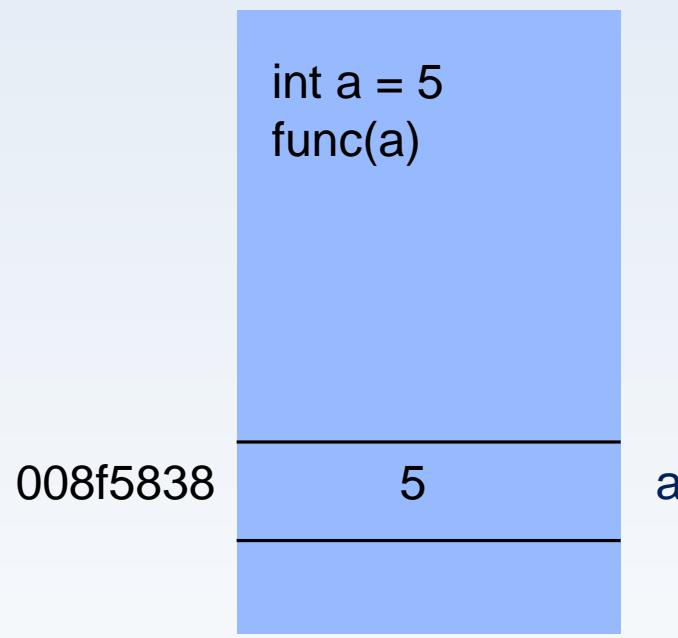
- `int i = 5;`
- `int *ptr=&i; //C style`
-  `int &j = i; //C++ style`
-
- `Integer i=new Integer(5); //Java Style`

Pass-by-Value vs. Pass-by-Reference

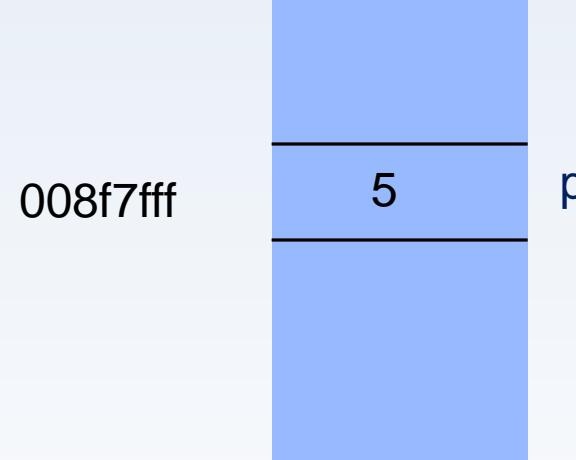
- When an argument is passed **pass-by-value**, a copy of the argument's value is made and passed to the called function.
- With **pass-by-reference**, the addresses of the parameters are passed to the function. Thus, the function is given direct access to these parameters.

Pass-by-Value

Main program



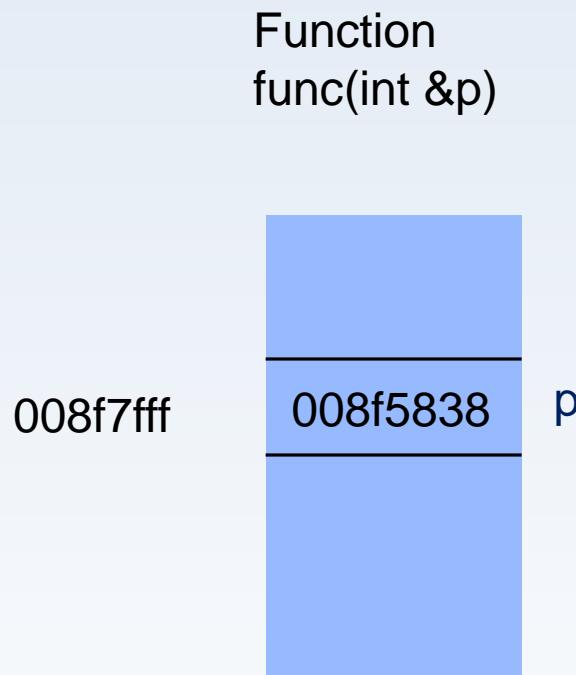
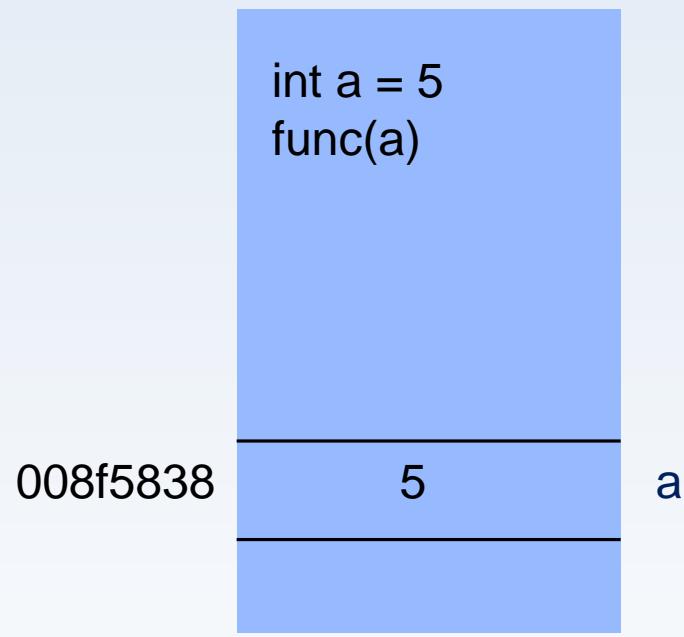
Function
func(int p)



When **p = 7** is executed in the function, the value of **p** changes, but the value of **a** does not

Pass-by-Reference

Main program



When $p = 7$ is executed in the function,
the value in address $\&p$ (that is, a) changes.

Arrays As Arguments

- When an array name is passed to a function, what is passed is the address of the first element of the array.
- Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address.
- **We can use this information as a pointer.**

Example

```
// return length of string s
int strlen(char *s){
    int n = 0;
    for (n = 0; *s != '\0'; s++)
        n++;
    return n;
}

int main()
{
    char a[10] = "array";
    int k;
    char *ptr = a;
    k = strlen("hello world");
    k = strlen(a);
    k = strlen(ptr);
    return 0;
}
```

example0.cpp

Example

All strings end with the null ('\0') character.
This value corresponds to "false".

```
int strlen(char *s){  
    int n = 0;  
    while (*s){  
        s++;  
        n++;  
    }  
    return n;  
}  
int main( )  
{    char a[10] = "array";  
    int k;  
    char *ptr = a;  
    k = strlen("hello world");  
    k = strlen(a);  
    k = strlen(ptr);  
    return 0;  
}
```

example0.cpp

Arrays as Arguments

- When an array name is passed to a function, the function can use this information as either an array or a pointer.
- It is also possible to pass part of an array to a function, by passing a pointer to the beginning of the subarray.

$f(&a[2]);$
 $f(a + 2);$

Both pass to the function f
the address of the subarray
beginning with the 3. element
of array a

Example

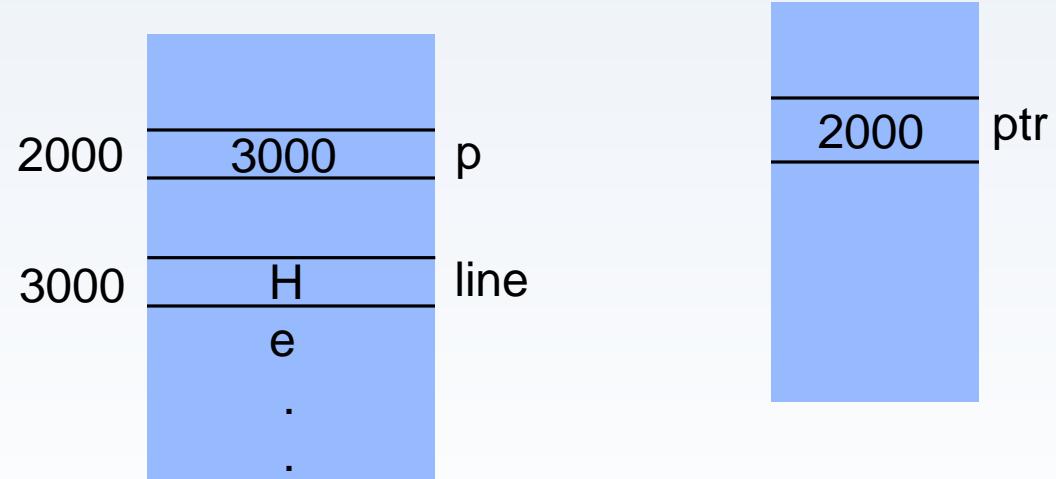
```
int strlen(char *s){  
    int n = 0;  
    for (n = 0; *s != '\0'; s++)  
        n++; for (n=0; s[n] != '\0'; n++);  
    return n;  
}  
int main( )  
{    char a[10] = "array";  
    int k;  
    char *ptr = a;  
    k = strlen("hello world");  
    k = strlen(&a[2]);  
    k = strlen(ptr);  
    return 0;  
}
```

Pointer to a Pointer as an Argument

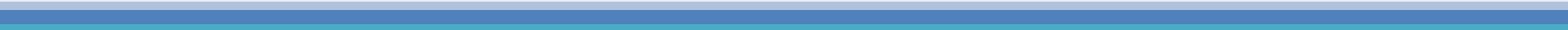
- If we want to change the value of an argument, we pass the address of the argument to the function.
- Similarly, if we want to change the value of a pointer, we pass the address of the pointer to the function (that is, we must pass a pointer to a pointer.)

Pointer to a Pointer as an Argument

```
void f(char **ptr){  
    (*ptr)++;  
}  
  
int main( )  
{    char line[20] = "Hello world";  
    char *p = line;  
    f(&p);  
    return 0;  
}
```



Dynamic Memory



Dynamic Memory Allocation

- At execution time, a program can obtain and use new memory space. Memory space obtained at execution time can be returned to the system when it is no longer needed.
- This ability is known as **dynamic memory allocation**.
- The limit on the size of the memory space that can be allocated is determined by how much the operating system allows. (You will learn about memory management in operating systems in courses to come.)
- In C, functions **malloc** and **free** are essential to manipulating dynamic memory space.
- In C++, the essential functions are the **new** and **delete** operators.
- In this course, we will use **new** and **delete**.

Pointers and Dynamic Memory Allocation

- It is necessary to use pointers to dynamically request memory space from the system.
- Since the program is compiled and running at that moment, it is not possible to give a symbolic name to the memory space to be allocated.

```
int integer1;  
int *dnumber;
```

- **integer1** is a symbolic name and the name of an integer. This name can be used where the variable is defined (in scope).
- **dnumber** is a pointer. It cannot be used yet because it does not point to any memory location. Pointers can only be used if they are pointing to a meaningful memory slot.
- To dynamically allocate memory, there has to be a pointer that can be used.

new Operator

```
dnumber = new int;
```

- This expression allocates memory space from the system for data of type `int` and assigns the address of this space to the `dnumber` pointer.
- The `new` operator both allocates memory space that is large enough to accommodate a variable of the right type and returns a pointer of a suitable type.
- Compared to the usage of `malloc`, this obviously provides a great convenience. The need for the `sizeof` call and the conversion of the returned "generic" pointer type is eliminated.
- If memory cannot be allocated for whatever reason, `new` returns `NULL`.
- Now, the `dnumber` pointer is pointing to a real location. This dynamic memory space can be accessed using this pointer.

```
*dnumber = 5;
```

delete Operator

- Dynamically allocated memory space should be returned to the system when it is no longer needed.
- Memory is returned using the expression

```
delete dnumber;
```

Note!

1. If new memory has not been allocated for a pointer that had the memory it was previously pointing to returned to the system, using **delete** again on this pointer may lead to unexpected errors.
2. **delete** can **only** be applied to memory space that has been **allocated with new**.
3. **delete** cannot be applied to pointers pointing to nondynamic variables. These memory slots cannot be returned.

Dynamic Creation of Arrays

- Dynamic memory may be allocated for the whole array at once.

```
float *darray;  
darray = new float[100];
```

- Here, an array large enough to hold 100 real variables has been dynamically created. The start address of the array has been assigned to the **darray** pointer.
- After the above assignment, **darray** can be used to access array elements.
- To return the array to the system:

```
delete [] darray;
```
- **Note:** When deallocating the array, we should not forget to use **[]**.

Discussion

- What happens if dynamically allocated memory slots are not returned?
- Try it out: What is the largest memory space that can be allocated at a time?

Abstract Data Types

Abstract Data Types - ADTs

- Integers, real numbers, and Booleans are built-in data types.
- Similarly, we could define more complex abstract data types such as the list, set, stack, and queue.
- Certain operations that can be performed are defined on every built-in data type.
 - Integer addition
 - Real number multiplication
 - Boolean "or" operation
- Likewise, abstract data types are incomplete without the operations that can be performed on them.

Example: On the set structure,

- We could define operations such as union, intersection, complement, and set size.

- **Basic idea:**
 - The implementation of these operations is written once in the program.
 - Any other part of the program that needs to perform an operation on the ADT can do so by calling the appropriate function.
 - If for some reason, implementation details need to be changed, it should be easy to do so by merely changing the functions that perform the ADT operations, and these changes should not affect the programs using this data type. This property is known as **transparency**.
- Which operations (functions) will be defined for an ADT is a design choice, and it is determined by the programmer depending on the need.
- How an ADT will be implemented (that is, how it will be designed and programmed) may vary from programmer to programmer. These design details and differences do not change the outside appearance of the ADT (as viewed by the programs that use it).

C++ and struct

- In C++, the structure provides a natural capsule for defining an abstract data type.
- Thus, the data type and the functions that define the operations to be performed on this type are located in the same capsule and logically related.

C

```
typedef struct DataType{  
    int a[10];  
    int elementnumber;  
} NewArrayType;  
  
int CountElmt(NewArrayType *d)  
{  
    return d->elementnumber;  
}
```

C++

```
struct NewArrayType{  
    int a[10];  
    int elementnumber;  
    int CountElmt();  
};  
  
int NewArrayType::CountElmt()  
{  
    return elementnumber;  
}
```

struct

- Structures cannot contain another structure of the same type. They can only contain a reference (pointer) of the same type.

```
struct NewArrayType{  
    int a[10];  
    int elementnumber;  
    NewArrayType *y;   
    int CountElmt();  
};
```

- Defining a structure does not reserve any space in memory. It only creates a new data type. However, when a structure variable (variable of the type the structure defines) is declared, memory is allocated for this variable.

struct

The following operations can be performed on structure variables:

- Assignment operation =
- Address operator &
- Member access operators . and ->
- sizeof operator

struct

- The members of a structure do not have to be located consecutively in memory.
- Passing structures to functions:
The whole structure or its members may be passed to the function. Structures are always passed pass-by-value.
- To pass a structure to a function by reference, we have to pass its reference.
- **Note:** An array that is a member of the structure also gets passed to the function by value.
- When the assignment operator (=) is used between structures, all members are copied.

struct (advanced topics)

- By defining nested structures, more general-purpose (easier to update based on the data type it holds) programs can be written.
- This situation is a choice completely dependent on the design.
- Example:

```
struct node{
    char firstname[20];
    char lastname[20];
    node *next;
};
void add(char *firstname, char *lastname){
    node *newnode = new node;
    strcpy(newnode->firstname, firstname);
    strcpy(newnode->lastname, lastname);
    . . .};
add("ahmet", "cetin");
```

Example

```
struct nodedata{  
    char firstname[20];  
    char lastname[20];  
};  
struct node{  
    nodedata data;  
    node *next;  
};  
void add(nodedata d){  
    node *newnode = new node;  
    newnode->data = d;  
    . . .};  
  
nodedata d = {"ahmet", "cetin"};  
add(d);
```

In structures, since the **“=”** operation **copies the whole content**, the data part will be completely copied independent of content. Thus, it will not be necessary to update the code by performing different copying operations for different types. The node “newnode” is stored in a different location in memory.

Example

- Note: What if the node data is as follows?

```
struct nodedata{  
    char firstname[20];  
    char lastname[20];  
};
```

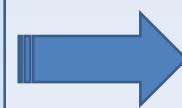


```
struct nodedata{  
    char *firstname;  
    char *lastname;  
};
```

`newnode->data = d;` Which operation will be performed?

The result of the operation in the first structure and the second structure will be different. In the first structure, array contents are copied to a new location, while in the second structure, only pointer values are copied. The values of `firstname` and `lastname` pointers will be copied. In other words, the address values they hold will be copied. In this case, the pointers in the `data` part of the "`newnode`" structure and pointers of the `d` structure will point to the same location in memory.

```
struct nodedata{
    char firstname[20];
    char lastname[20];
};
```



```
struct nodedata{
    char *firstname;
    char *lastname;
};
```

Name	Value
v	{...}
+ name	0x0012fee4 "ahmet"
+ lastname	0x0012fef8 "cetin"
newnode	0x00441ba0
data	{...}
+ name	0x00441ba0 "ahmet"
+ lastname	0x00441bb4 "cetin"
next	0xcdcdcdcd

Name	Value
v	{...}
+ name	0x0042e024 "ahmet"
+ lastname	0x0042e01c "cetin"
newnode	0x00441bc0
data	{...}
+ name	0x0042e024 "ahmet"
+ lastname	0x0042e01c "cetin"
next	0xcdcdcdcd

To do: In recitation

- Solution to Homework 2
- Function calls
- Parameter passing
- Where in a program the variable may be used (scope)

Homework

In the PhoneBook example, phonerecords were defined as constant-size arrays. Realize the necessary operations (in code) for defining dynamic sizes for these arrays.

Homework

The phonebook example was solved using an array.

In this homework, you should use a dynamic array instead of a static array.

In this structure, this is what you need to do:

First, inside the phonebook structure, you should define a variable `size=5`.

Within `Book.create`, an array of this size will be dynamically allocated.

Then, every addition operation will check to see if the current index has reached `size`. If the array is full (that is, the index has reached `size`), then a function called `increase size` will be called. This function will allocate a dynamic array of `size*2` from the memory. We will call this `newarray`. This function will first copy the elements in the array to the beginning of the `newarray` using a loop. Then, the old array will be returned to the system. The array will be assigned to the new array. And the new size will be defined as two times as big.

Data Structures

Linked List

The List Abstract Data Structure

- A list is made of the same kind of elements.
- Elements in the list:
 - A_1, A_2, \dots, A_N
 - A_1 : First element in the list
 - A_i : i th element in the list
- The list size is variable.
- The primary operations for the list:
 - `printList`: Print list elements to the screen.
 - `makeEmpty`: Delete all elements in the list.
 - `find`: Search for a given value in the list, find the related element.
 - `insert`: Insert data to the appropriate place in the list.
 - Insert to the beginning
 - Insert to the end
 - Insert to a particular position
 - `remove`: Find data and remove it from the list.

Example

List: 34, 12, 52, 16, 12

- `find(52)` Returns the number 3 which is the order of 52
- `insert(X, 3)` After this operation, the list looks like:
34, 12, X, 52, 16, 12
- `remove(52)` After this operation, the list looks like:
34, 12, X, 16, 12

- Operations and the functions that realize these operations could have a great variety.
- The inputs and outputs of every operation could be defined in different ways.

- `insert(X,3)` Add X as the 3. element.
- `insert(X, 12)` Add X after data 12.
- `insert(X)` Add X to the list in order.

List ADT with Array Implementation

- All operations we have defined can be realized using the array structure of C++.
- **Problem: The array size has to be known in advance.**
 1. The array could be defined to have a constant size at the beginning of the program.
`int A[100];`
 2. It could be defined to have a size given during execution.
`int arraysize;
int *A;
cin << arraysize;
A = new int[arraysize];`

- In both cases, the number of elements to be placed in the list has to be known in advance.
- If defined to be unnecessarily big → waste of memory space
- If defined to be too small → insufficient space

Importance of the Dynamic Structure

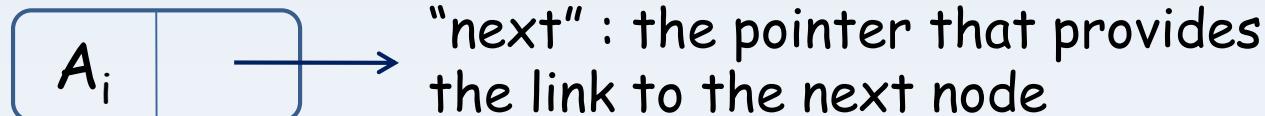
- `printList` and `find` are **linear** operations.
 - List elements are traversed in order from beginning to end.
- However, `insert` and `remove` are **expensive** operations to perform on the array.
 - To insert, all elements to the right of the inserted element should be shifted once to the right.
 - When the element is removed, the elements on the right should be shifted once to the left to get rid of the gap.

Expensive = Too many element reads/writes (swaps) performed.

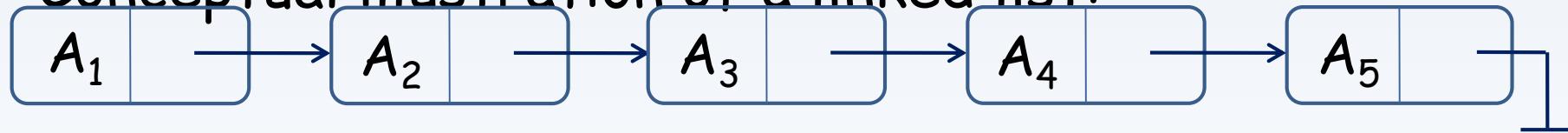
- Since the list structure requires these types of operations and can have variable size, using arrays to implement list structures is not preferred.

Linked List Implementation of the List ADT

- The linked list is made up of connected units.
- The units are called **nodes**.
- The type of each node is a structure made up of several fields; that is, it is a record.



- Conceptual illustration of a linked list:



- The value of the next pointer of the last node is **NULL**.

Linked List Implementation of the List ADT

- The nodes of the linked list **are not stored contiguously in memory!**
 - As opposed to an array.



- To access a list, we have to know the address of the first node.
 - This address is 1000 above.
 - A special pointer is used as **the list head pointer**.

Phone Book

- We will see how the list operations can be performed on the phone book example introduced in the previous weeks.
- First, we have to make some changes in the program:
- At the beginning of `phoneprog.cpp`

```
#include "list.h"
using namespace std;

typedef Phone_node Phone_Record;
typedef List Datastructure;

Datastructure book;
The data structure will
be List.
```

→ Header file that contains the structure where the list structure is defined

→ The nodes of the list structure are defined as `Phone_node`. The previous main program block uses `Phone_Record`. `Phone_node` will be used with the name `Phone_Record`.

Changes to the Example

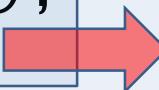
- A new option for clearing the list was added to the menu.

```
void print_menu(){  
    cout << endl << endl;  
    cout << "Phone Book Application" << endl;  
    cout << "Select an operation" << endl;  
    cout << "S: Record Search" << endl;  
    cout << "A: Record Add" << endl;  
    cout << "U: Record Update" << endl;  
    cout << "D: Record Delete" << endl;  
    cout << "C: Delete All" << endl;  
    cout << "E: Exit" << endl;  
    cout << endl;  
    cout << "Enter an option {S, A, U, D, C, E} : ";  
}
```

Functions

- No changes were made to the functions called by the main function:

```
void search_record();  
void add_record();  
void delete_record();  
void update_record();  
void clear_list();
```



New function

- In the phone book, the interfaces of the functions that enable operations also stayed the same, but of course their bodies will change.

```
void create();  
void close();  
void makeEmpty();  New function  
void insert(Phone_node *);  
void remove(int ordernum);  
int search(char *);  
void update(int recordnum, Phone_node *);
```

New Record Structure

```
#define NAME_LENGTH 30
#define PHONENUM_LENGTH 15
```

```
struct Phone_node{
    char name[NAME_LENGTH];
    char phonenum[PHONENUM_LENGTH];
    Phone_node *next;
};
```

- We have added the next pointer field that list nodes should have.

List Structure

```
#ifndef LIST_H
#define LIST_H
#include "node.h"

struct List{
    Phone_node *head;
    int nodecount;
    void create();
    void close();
    void printList();
    void makeEmpty();
    void insert(Phone_node *);
    void remove(int ordernum);
    int search(char *);
    void update(int recordnum, Phone_node *);
};

#endif
```

create()

```
void List::create (){  
    head = NULL;  
    nodecount = 0;  
}
```

- Before starting to use the List structure, we should first initialize it.

```
typedef Phone_node Phone_Record;  
typedef List Datastructure;
```

```
Datastructure book;  
int main(){  
    book.create();  
    ...
```

```
int List::search(char *target){
    Phone_node *traverse;
    int counter = 0;
    int found = 0;
    traverse = head;
    bool all = false;
    if ( strcmp(target, "*") == 0 )
        all = true;

    while (traverse){
        counter++;
        if (all){
            cout << counter << "." << traverse->name << " " <<traverse->phonenum
                                         <<endl;
            found++;
        }
        else if (strnicmp(traverse->name, target, strlen(target)) == 0){
            found++;
            cout << counter << "." << traverse->name << " " <<traverse->phonenum
                                         <<endl;
        }
        traverse = traverse->next;
    }
    return found;
}
```

```
Phone Book Application
Choose an operation
S: Record Search
A: Record Add
U: Record Update
D: Record Delete
C: Delete All
E: Exit
```

```
Enter a choice {S, A, U, D, C, E} : s
```

```
Please enter the name of the person you want to search for (press '*' for full list):
```

```
*
```

```
1. Gülsen 5324444444
```

```
2. Veli 2123333333
```

remove()

```
void List::remove(int
ordernum){
Phone_node *traverse, *tail;
int counter = 1;
traverse = head;
if (ordernum <= 0){
    cout << "Invalid record
          order number.\n";
    return;
}
if (ordernum == 1){
    head = head->next;
    delete traverse;
    nodecount--;
    return;
}
```

```
while ((traverse != NULL) &&
       (counter < ordernum)){
    tail = traverse;
    traverse = traverse->next;
    counter++;
}
if (counter < ordernum){
// given order num too large
    cout << "Could not find
          record to delete.\n";
}
else{ // record found
    tail->next = traverse->next;
    delete traverse;
    nodecount--;
}
```

makeEmpty()

```
void List::makeEmpty(){
    Phone_node *p;
    while (head){
        p = head;
        head = head->next;
        delete p;
    }
    nodecount = 0;
}
```

insert()

```
void List::insert(Phone_node
                  *toadd){
    Phone_node *traverse, *tail;
    Phone_node *newnode;
    traverse = head;
    newnode = new Phone_node;
    *newnode = *toadd;
    newnode->next = NULL;
    if (head == NULL){
        //first node being added
        head = newnode;
        nodecount++;
        return;
    }
    if (strcmp(newnode->name, head->name) < 0){
        //Insert to head of list
        newnode->next = head;
        head = newnode;
        nodecount++;
        return;
    }
    while (traverse &&
           (strcmp(newnode->name, traverse->name) > 0)){
        tail = traverse;
        traverse = traverse->next;
    }
    if (traverse){ // Insert into a position
        newnode->next = traverse;
        tail->next = newnode;
    }
    else // Insert to end
        tail->next = newnode;
    nodecount++;
}
```

update()

```
void List::update(int recordnum, Phone_node *newnode){  
    Phone_node *traverse;  
    int counter = 1;  
    traverse = head;  
    while (traverse && (counter < recordnum)){  
        counter++;  
        traverse = traverse->next;  
    }  
    if (traverse){  
        newnode->next = traverse->next;  
        *traverse = *newnode;  
    }  
    else  
        cout << "Invalid number for record to be  
        updated.\n";  
}
```

End of Program

- When the program is being ended, all the space allocated for dynamic data structures has to be returned to the system.
- The records in the phone book are held in a linked list.
- When the program is ending, all nodes must be deleted.

```
int main(){
    book.create();
    bool end = false;
    char choice;
    while (!end) {
        print_menu();
        cin >> choice;
        end = perform_operation(choice);
    }
    book.close();
    return EXIT_SUCCESS;
}
```

`void List::close(){
 makeEmpty();
}`

Making Data Permanent

- After the program has been closed, the data has to be stored in the hard disk so that it is not lost.
- That is why the records are saved to a file when closing the program in our lecture example.

```
struct List{  
    ...  
    char *filename;  
    FILE *phonebook;  
    void read_fromfile();  
    void write_tofile();  
};
```

```
void List::create(){
    head = NULL;
    nodecount = 0;
    read_fromfile();
}
```

```
void List::close(){
    write_tofile();
    makeEmpty();
}
```

```

void List::read_fromfile(){
    struct File_Record{
        char name[NAME_LENGTH];
        char phonenum[PHONENUM_LENGTH];
    };
    File_Record record;
    Phone_node *newnode;
    filename = "phonebook.txt";
    if ( !(phonebook = fopen( filename, "r+" ) ) )
        if ( !(phonebook = fopen( filename, "w+" ) ) ){
            cerr << "File could not be opened" << endl;
            exit(1);
        }
    fseek(phonebook, 0, SEEK_SET);
    while ( !feof(phonebook) ){
        newnode = new Phone_node;
        fread( &record, sizeof (File_Record), 1, phonebook );
        if (feof(phonebook)) break;
        strcpy(newnode->name, record.name);
        strcpy(newnode->phonenum, record.phonenum);
        newnode->next = NULL;
        insert(newnode);
        delete newnode;
    }
    fclose(phonebook);
}

```

```
void List::writeToFile(){
    struct File_Record{
        char name[NAME_LENGTH];
        char phonenum[PHONENUM_LENGTH];
    };
    File_Record record;
    Phone_node *p;
    if ( !(phonebook = fopen( filename, "w+" ) ) ){
        cerr << "File could not be opened" << endl;
        exit(1);
    }
    p = head;
    while (p){
        strcpy(record.name, p->name);
        strcpy(record.phonenum, p->phonenum);
        fwrite(&record,sizeof(File_Record), 1, phonebook);
        p = p->next;
    }
    fclose(phonebook);
}
```

Data Structures

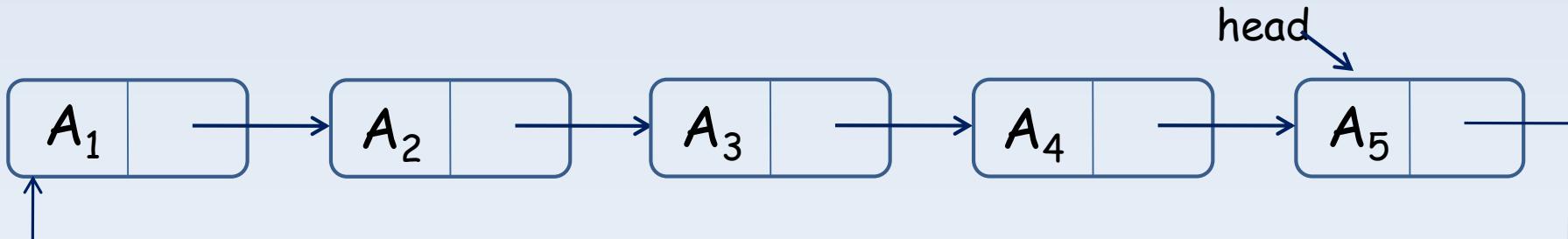
Implementations and Types of Lists

Circular List

Circular List

- The circular list is a widely used list implementation.
- A link is established from the last node to the first node.
- The “next” field of the last node is not NULL, but instead points to the first node.
- Advantage:
 - Starting from any node in the list, we can traverse to the end of the list and get back to the beginning.
- We do not need to make any changes in the definitions (the node and list structures).
- How we determine that we have reached the end of the list will change.
- In general, the bodies of functions such as `insert()` and `remove()` that perform list operations will change.

Circular List

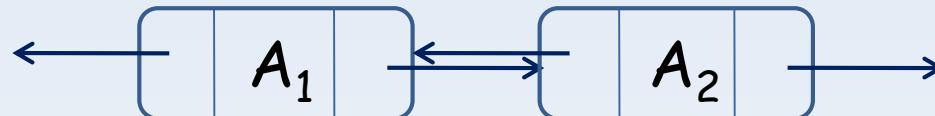


- In a circular list, the list head pointer must point to the end of the list.
 - Thus, both the beginning of the list and the end of the list can be reached in one step.
 - **head** points to the last node in the list.
 - **head->next** points to the first node in the list.
 - This makes it easier to insert to and remove from the beginning and to insert to the end.

Doubly Linked List

Doubly Linked List

- List nodes contain both forward and backward links.



- The list can be traversed in both directions by following the pointers.
- We have to make changes to the list operations.
- We will make the following changes to the design of the list:
 - The node structure will change.
 - The list structure will contain both head and tail pointers.
 - The bodies of list operations will change.

Node Structure

```
struct Phone_node{  
    char name[NAME_LENGTH];  
    char phonenum[PHONENUM_LENGTH];  
    Phone_node *next;  
    Phone_node *previous;  
};
```

List Structure

```
struct List{  
    Phone_node *head, *tail;  
    int nodecount;  
    char *filename;  
    FILE *phonebook;  
    void create();  
    void close();  
    ...  
};
```

create()

```
void List::create(){
    head = NULL;
    tail = NULL;
    nodecount = 0;
    read_fromfile();
}
```

insert()

```
void List::insert(Phone_node *toadd){  
    Phone_node *traverse, *newnode;  
    traverse = head;  
    newnode = new Phone_node;  
    strcpy(newnode->name, toadd->name);  
    strcpy(newnode->phonenum, toadd->phonenum);  
    newnode->previous = NULL;  
    newnode->next = NULL;  
    if (head == NULL) { // first node is being added  
        head = newnode;  
        tail = newnode;  
        nodecount++;  
        return;  
    }  
    if (strcmp(newnode->name, head->name) < 0) { // add to head of list  
        newnode->next = head;  
        head = newnode;  
        (newnode->next)->previous = newnode;  
        nodecount++;  
        return;  
    }  
}
```

insert() (continued)

```
•     while ( traverse && (strcmp(newnode->name, traverse->name) > 0) ) {  
•             // newnode's name comes after traverse's  
•             traverse = traverse->next;  
•     }  
•     if (traverse) { // insert in between  
•         newnode->next = traverse;  
•         newnode->previous = traverse->previous;  
•         (traverse->previous)->next = newnode;  
•         traverse->previous = newnode;  
•     }  
•     else{ // insert to end  
•         tail->next = newnode;  
•         newnode->previous = tail;  
•         tail = newnode;  
•     }  
•     nodecount++;  
}
```

remove()

```
void List::remove(int ordernum){  
    Phone_node *traverse;  
    int counter = 1;  
    traverse = head;  
  
    if (ordernum <= 0) {  
        cout << "Invalid record order number.\n";  
        return;  
    }  
  
    if (ordernum == 1){  
        head = head->next;  
        head->previous = NULL;  
        delete traverse;  
        nodecount--;  
        return;  
    }  
}
```

remove() (continued)

```
while ( (traverse != NULL) && (counter < ordernum) ) {  
    traverse = traverse->next;  
    counter++;  
}  
if (counter < ordernum) { // given record num too large  
    cout << "Could not find record to delete.\n";  
}  
else { //record found  
    (traverse->previous)->next = traverse->next;  
    if (traverse->next)  
        traverse->next->previous = traverse->previous;  
    else  
        tail = traverse->previous;  
    delete traverse;  
    nodecount--;  
}
```

search() and update()

- No changes have to be made to these functions.
- Since the search() function starts from the head of the list and searches the list going forward, previous fields are not used.
- Same is true for update().

Multilist Example

Designing the Data to Suit the Program

- Many variations could be created on the linked list basic structure.
- The programmer must determine the most suitable structure for the program at hand.
- When the suitable structure is selected, the best data storage and access environment has been prepared for writing the program.
- Before starting to write a program, the data must be designed carefully.
- Data is the fundamental building block of a program. When data is designed correctly, writing, debugging, and testing the program becomes easier.

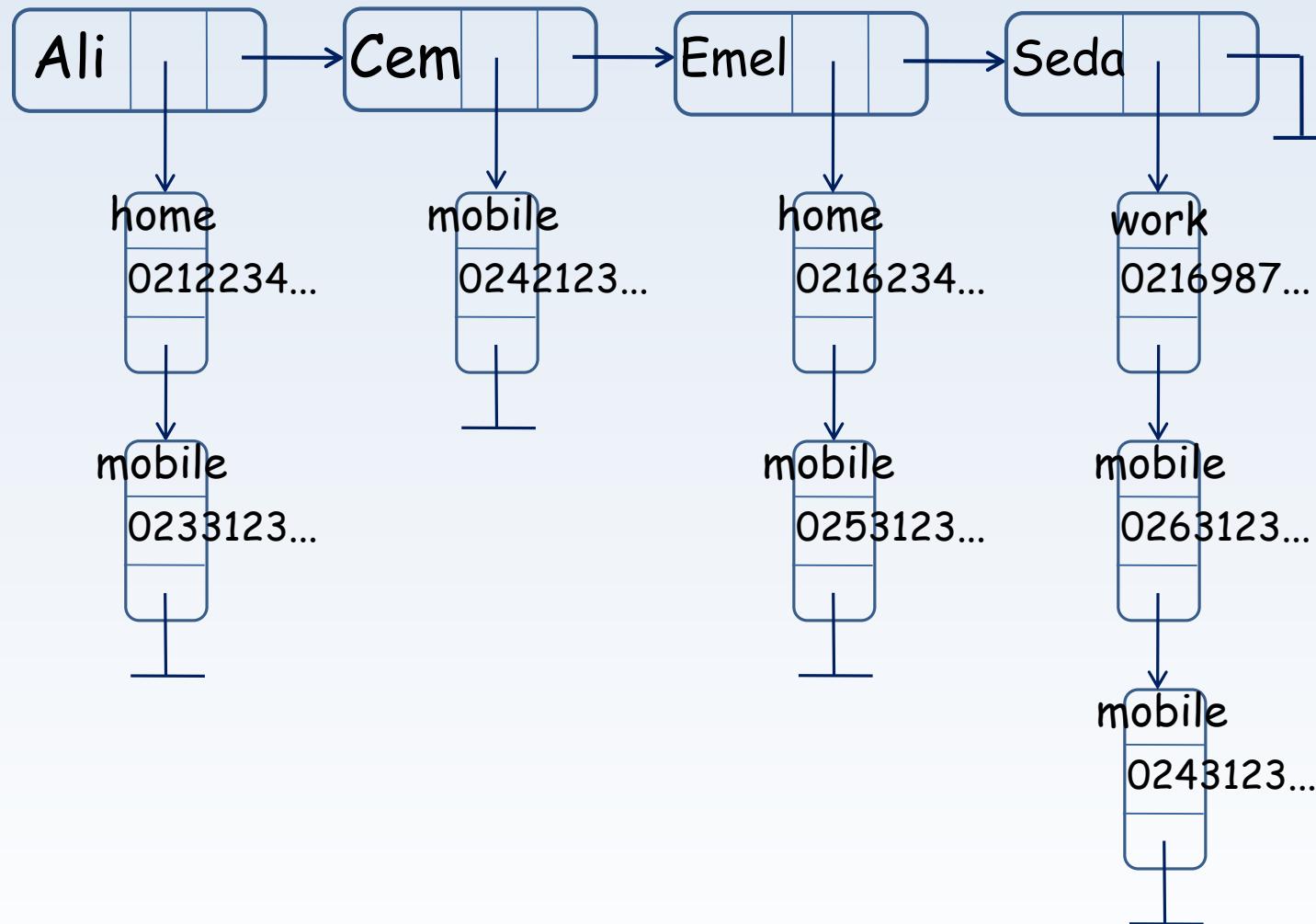
Types of Linked Lists

- The linked structure could be used for creating multidimensional lists: **Multilist**
- In multilists, more than one list is constructed using different node types.
- The data and pointer types to be contained in each node type are designed based on the structure.

A New Design for the Phone Book

- Our phone book example actually does not take into account the case of each person having more than one phone number.
- Solution:
 - The main list will be made up of nodes that hold the name of a person and a pointer to the list that contains the phone numbers of that person.
 - Phone numbers belonging to each person will be kept in a separate list along with phone types (work, home, mobile).
- We are designing a two-dimensional structure.

Multilist Structure for Phone Book



Node Structures

```
#define NAME_LENGTH 30
#define PHONENUM_LENGTH 15
#define TNAME_LENGTH 4

struct number{
    char type[TNAME_LENGTH];
    char num[PHONENUM_LENGTH];
    number *next;
};

struct Phone_node{
    char name[NAME_LENGTH];
    number *phonenum;
    Phone_node *next;
};
```

List Structure

```
struct List{  
    Phone_node *head;  
    int personcount;  
    char *filename;  
    FILE *phonebook;  
    void create();  
    Phone_node *create_node(char *, char *, char *);  
    void close();  
    void makeEmpty();  
    void insert(char *, char *, char *);  
    void remove(int ordernum);  
    int search(char *);  
    void update(int, char *);  
    void read_fromfile();  
    void write_tofile();  
};
```

Changes in declaration and body
Changes in body

makeEmpty()

```
void List::makeEmpty()
{
    Phone_node *p;
    number *q;
    while (head) {
        p = head;
        head = head->next;
        q = p->phonenum;
        while (q) {
            p->phonenum = p->phonenum->next;
            delete q;
            q = p->phonenum;
        }
        delete p;
    }
    personcount = 0;
}
```

create_node()

```
Phone_node * List::create_node(char *name, char *phone,
                               char *type){  
    Phone_node *newperson;  
    number *newnum;  
    newperson = new Phone_node;  
    strcpy(newperson->name, name);  
    newnum = new number;  
    newperson->phonenum = newnum;  
    strcpy(newnum->num, phone);  
    strcpy(newnum->type, type);  
    newnum->next = NULL;  
    newperson->next = NULL;  
    return newperson;  
}
```

insert()

```
void List::insert(char *newname, char *newphone, char *newtype){  
    Phone_node *traverse, *behind, *newperson;  
    number *newnum;  
    traverse = head;  
    if (head == NULL) { // first node being added  
        head = create_node(newname, newphone, newtype);  
        personcount++;  
        return;  
    }  
    if ( strcmp(newname, head->name) < 0 ) { // add to head of list  
        newperson = create_node(newname, newphone, newtype);  
        newperson->next = head;  
        head = newperson;  
        personcount++;  
        return;  
    }  
}
```

insert() (continued)

```
while ( traverse && (strcmp(newname, traverse->name) > 0) ) {  
    // newname comes after traverse's name  
    behind = traverse;  
    traverse = traverse->next;  
}  
if (traverse && strcmp(newname, traverse->name) == 0){  
    // this name was added before; just add phone number  
    newnum = new number;  
    newnum->next = traverse->phonenum;  
    traverse->phonenum = newnum;  
    strcpy(newnum->num, newphone);  
    strcpy(newnum->type, newtype);  
}  
else {  
    newperson = create_node(newname, newphone, newtype);  
    if (traverse) { // inserting new name in between  
        newperson->next = traverse;  
        behind->next = newperson;  
    }  
    else // insert to the end  
        behind->next = newperson;  
    personcount++;  
}  
}
```

remove()

```
void List::remove(int ordernum){  
    Phone_node *traverse, *behind;  
    number *pn;  
    int counter = 1;  
    traverse = head;  
    if (ordernum <= 0) {  
        cout << "Invalid record order number.\n";  
        return;  
    }  
    if (ordernum == 1) {  
        head = head->next;  
        pn = traverse->phonenum;  
        while (pn) {  
            traverse->phonenum = pn->next;  
            delete pn;  
            pn = traverse->phonenum;  
        }  
        delete traverse;  
        personcount--;  
        return;  
    }  
}
```

remove() (continued)

```
while ( (traverse != NULL) && (counter < ordernum) ){
    behind = traverse;
    traverse = traverse->next;
    counter++;
}
if (counter < ordernum){ // given record num too large
    cout << "Could not find record to delete.\n";
}
else{ // record found
    behind->next = traverse->next;
    pn = traverse->phonenum;
    while (pn) {
        traverse->phonenum = pn->next;
        delete pn;
        pn = traverse->phonenum;
    }
    delete traverse;
    personcount--;
}
}
```

search()

```
int List::search(char *target){  
    Phone_node *traverse;  
    number *pn;  
    int counter = 0;  
    int found = 0;  
    traverse = head;  
    bool all = false;  
  
    if (target[0] == '*')  
        all = true;
```

search() (continued)

```
while (traverse) {
    counter++;
    if (all) {
        cout << counter << "." << traverse->name << endl;
        pn = traverse->phonenum;
        while (pn) {
            cout << pn->type << " : " << pn->num << endl;
            pn = pn->next;
        }
        found++;
    }
    else if ( strncmp(target, traverse->name, strlen(target)) == 0 ) {
        found++;
        cout << counter << ". record:" << traverse->name << endl;
        pn = traverse->phonenum;
        while (pn) {
            cout << pn->type << " : " << pn->num << endl;
            pn = pn->next;
        }
    }
    traverse = traverse->next;
}
return found;
}
```

update()

- Updates only the name.

```
void List::update(int recordnum, char *newname){  
    Phone_node *traverse;  
    int counter = 1;  
    traverse = head;  
    while (traverse && (counter < recordnum) ){  
        counter++;  
        traverse = traverse->next;  
    }  
    if (traverse)  
        strcpy(traverse->name, newname);  
    else  
        cout << "Invalid record number to update.\n";  
}
```

read_fromfile()

```
void List::read_fromfile(){
    struct File_Record{
        char name[NAME_LENGTH];
        char phonenum[PHONENUM_LENGTH];
        char type[TNAME_LENGTH];
    };
    File_Record record;
    filename = "phonebook.txt";
    if ( !(phonebook = fopen( filename, "r+" ) ) ) {
        if ( !(phonebook = fopen( filename, "w+" ) ) ) {
            cerr << "File could not be opened." << endl;
            cerr << "will work in memory only." << endl;
            return;
        }
    }
    fseek(phonebook, 0, SEEK_SET);
    while ( !feof(phonebook) ) {
        fread(&record, sizeof(File_Record), 1, phonebook);
        if ( feof(phonebook) ) break;
        insert(record.name, record.phonenum, record.type);
    }
    fclose(phonebook);
}
```

writeToFile()

```
void List::writeToFile(){
    struct File_Record{
        char name[NAME_LENGTH];
        char phonenum[PHONENUM_LENGTH];
        char type[TNAME_LENGTH];
    };
    File_Record record;
    Phone_node *names;
    number *n;
    if(!phonebook =
        fopen( filename, "w+" ) ) ){
        cerr << "File could not be
                  opened"
        << endl;
        return;
    }
    names = head;
    while (names){
        n = names->phonenum;
        while (n){
            strcpy(record.name, names->name);
            strcpy(record.phonenum, n->num);
            strcpy(record.type, n->type);
            fwrite(
                &record, sizeof(File_record),
                1, phonebook);
            n = n->next;
        }
        names = names->next;
    }
    fclose(phonebook);
}
```

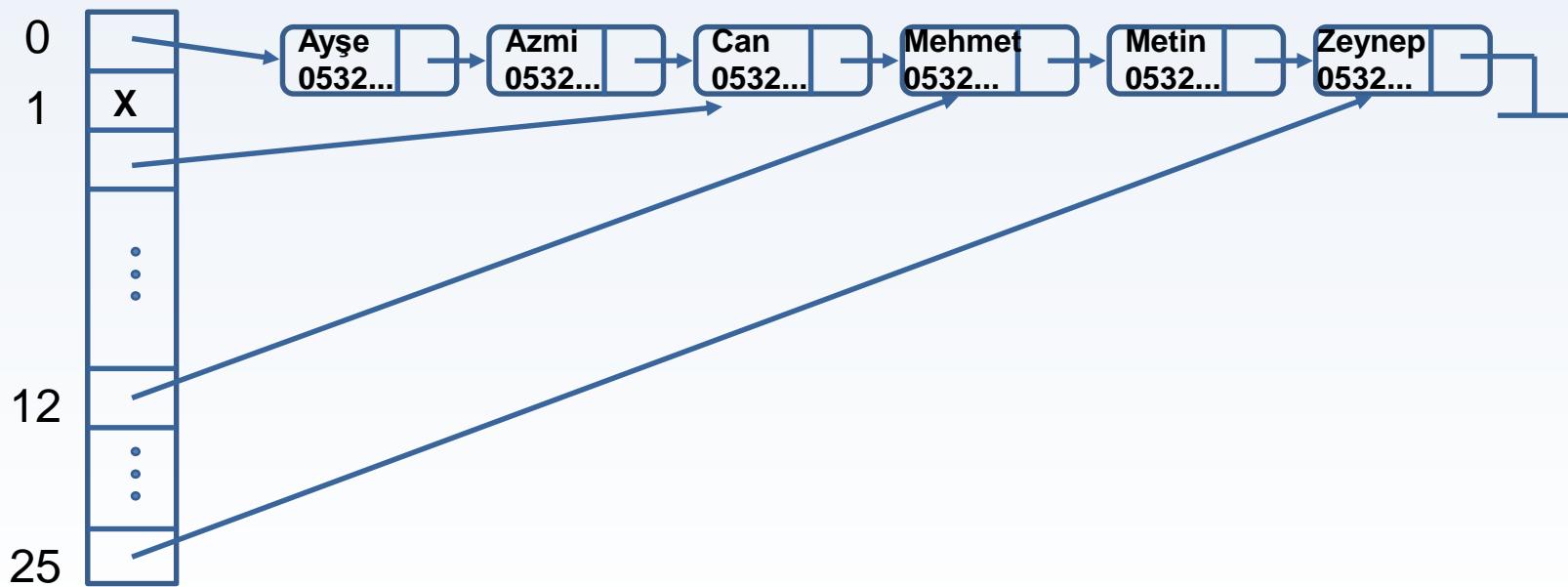
Data Structures

List Applications

List Application– Lecture Example

- Searching in a linked list can become a very time consuming operation when there are many list elements.
- For example, in an ordered phone book containing 3000 phone number records, searching for the record "Zeynep Ata" may be a very costly operation.
- The search function given in the lecture example operates by traversing the list records from the very beginning and hence is an expensive method.

- To speed up the search operation and access to data, we could design a new data structure
- In this data structure, letter indices (corresponding to their order in the alphabet) are held in an array.
- Thus, faster access to records starting with the same letter could be provided.
- For example, the records starting with letter 'M' could be accessed using the 13. element of the array (array[12]).



This index array is defined as follows:

```
#ifndef LIST_H
#define LIST_H
#include "node.h"

struct List{
    Phone_node *head;
    Phone_node *index[26];
    int nodecount;
    char *filename;
    FILE *phonebook;
    void create();
    void close();
    void makeEmpty();
    void insert(Phone_node *);
    void remove(int ordernum);
    int search(char *);
    void update(int recordnum, Phone_node *);
    void read_fromfile();
    void write_tofile();
};

#endif
```

Adding

- The function for adding to a linked list should be written in such a way that it also updates the index array mentioned above.
- **Note:** There will be no uppercase/lowercase distinction ("Ahmet" and "ahmet" will be located next to each other in the list).
- We assume that names start with only the letters of the alphabet (a-z, A-Z).
- **Hint 1:** We can ignore the case of names beginning with special Turkish characters and use their ASCII values.
- **Hint 2:** The function "int tolower(int)" returns the lowercase version of a given character.

```

• void List::insert(Phone_node *toadd){
•
•     Phone_node *traverse, *behind, *newnode;
•
•     traverse = head;                                // add to head of list
•     newnode = new Phone_node;                      if (strcmp(newnode->name, head->name) < 0){
•
•     *newnode = *toadd;                            newnode->next = head;
•
•     newnode->next = NULL;                         head = newnode;
•
•     int ch = tolower(newnode->name[0]);          nodecount++;
•
•     index[ch - 'a'] = newnode;                   return;
•
• }
•
• // first node being added
•
•     if (head == NULL) {
•
•         head = newnode;
•
•         nodecount++;
•
•         index[ch - 'a'] = newnode;
•
•         return;
•
• }
•
•
•     while (traverse &&
•
•             (strcmp(newnode->name, traverse->name) > 0)){
•
•         behind = traverse;
•
•         traverse = traverse->next;
•
•     }
•
•
•     newnode->next = traverse;
•     behind->next = newnode;
•
•     if (tolower(behind->name[0]) != tolower(newnode->name[0]))
•
•         index[ch - 'a'] = newnode;
•
•     nodecount++;
•
• }
•
•
• }

```

Searching

- The function for searching in a linked list is written in a way to conduct fast search using the data structure above.
- The parameters that will be passed to this function can only start with the letters of the alphabet (a-z, A-Z).
- The parameter could be a single letter or several letters.
- Similar to the search function created in class, when a single letter is entered, all records starting with that letter will be displayed.
- The function will be written to make the fewest possible number of comparisons.

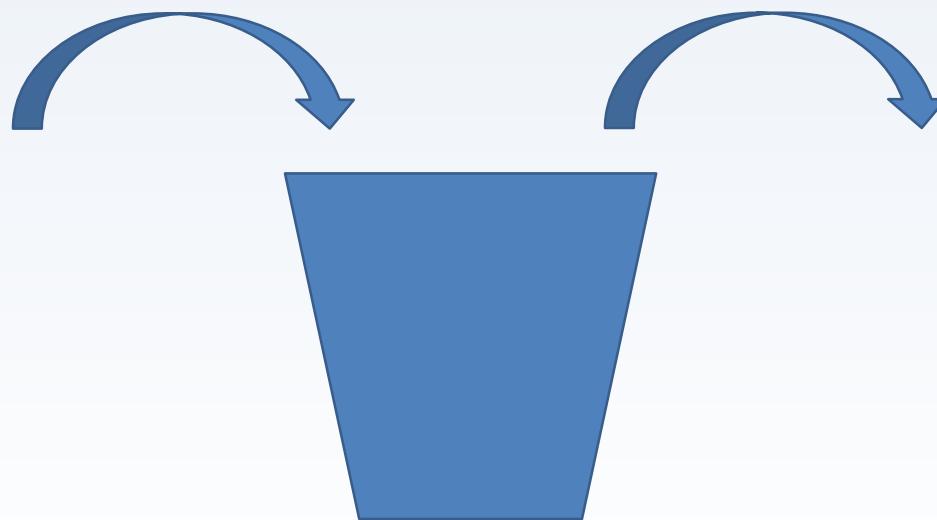
```
• int List::search(char *target)
• {
•     Phone_node *traverse;
•     int counter = 0;
•     int found = 0;
•     traverse = index[target[0] - 'a'];
•
•     while ( traverse && tolower(traverse->name[0]) == tolower(target[0]) ) {
•         counter++;
•
•         if ( strnicmp(traverse->name, target, strlen(target)) == 0 ) {
•             found++;
•             cout << counter << ". " << traverse->name << " " << traverse->phonenum <<
•         endl;
•     }
•     traverse = traverse->next;
• }
• return found;
• }
```

Data Structures

Stack

Stack

- The stack is the simplest of data structures.
- Placing something onto the stack and removing something from the stack is restricted to the top of the stack.
 - That is why it is also defined as **Last In First Out (LIFO)** storage.



- Stacks are used when nested blocks, local variables, recursive definitions, and backtracking operations are needed in programming.
- A typical programming example where stacks are used is the processing of arithmetic terms (containing parentheses and operator precedence).
- Another example could be the path-finding problem in a labyrinth that necessitates backtracking.

Stack Operations

- **pushing**: operation of adding a new element onto the top of the stack. The added element becomes the topmost element in the stack.

push(...)

- **popping**: operation of pulling the topmost element from the stack.

pop()

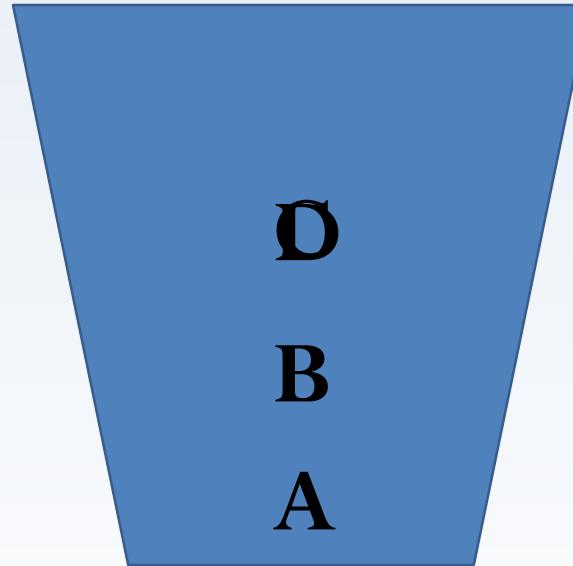
- **checking emptiness**: operation of checking if the stack is empty.

isempty()

Exercise

Perform these operations in order on a stack:

- push('A');
- push('B');
- push('C');
- pop();
- push('D');
- pop();
- pop();



Example: Arithmetic operations

- Checking for balanced parentheses in an arithmetic expression:

$$7 - ((x^*((x+y)/(y-3))+y)/4)$$

- Constraints:
 - There must be an equal number of right and left parentheses
 - A left parenthesis must correspond to each right parenthesis
- Solution
 1. Every time left parenthesis encountered, push onto stack.
 2. Every time right parenthesis encountered, pop from stack. Error if no popped element.
 3. When the end of the expression has been reached, error if the stack is not empty.

Example

- For checking correct usage of parentheses
- The system accepts three kinds of parentheses:
(), {}, []
- On each line of input, for each type of left parenthesis, we will test the presence of a right parenthesis of the same kind and their appearance in the right order.

Valid

- ()
- {}[]
- ([[]])
- [{()}{()}]]
- ()()()

Invalid

-)
- [
- {]
-))))()

Solution

For every c character in the input line

if c is a left_symbol → push onto stack.

if c is a right_symbol →

if stack empty → error "left symbol missing"

else, pop an s symbol from the stack

if s and c not compatible → error

"ordering not appropriate"

If stack not emptied → error "missing right symbol"

Example: Evaluation of Arithmetic Expressions

- $X \leftarrow ((A/(B^{**}C)) + (D^*E)) - (A^*C)$
- operands: A, B, C, D, E
- operators: /, **, +, -, *
- Each operation has a certain precedence. The use of parentheses affects the result. Using parentheses, different results could be obtained
- While generating code, compilers first convert the given expression to a representation called **postfix**. This “postfix” expression is then processed using the stack structure. In this representation, the operands are written before the operators (A B +). When the operation to be processed (+) is popped from the stack, the number of operands this operation requires is popped from the stack (+ operator requires 2 operands).

Example: Call Stack

- Operating systems push the current state of the code and data into a stack before branching out into each function.
- Thus, when it returns to the branching point, it has returned to the state at the time of branching.

Stack Data Structure

```
#ifndef STACK_H
#define STACK_H
#define STACKSIZE 5
typedef char StackDataType;
struct Stack{
    StackDataType element[STACKSIZE];
    int top;

    void create();
    void close();
    bool push(StackDataType);
    StackDataType pop();
    bool isempty();
};

#endif
```

```

void Stack::create(){
    top = 0;
}
void Stack::close(){
}
bool Stack::push(StackDataType newElement){
    if (top < STACKSIZE) {
        element[top++] = newElement;
        return true;
    }
    return false;
}
StackDataType Stack::pop(){
    return element[--top];
}
bool Stack::isempty(){
    return (top == 0);
}

```

Had the stack data type not been a simple character, but a struct structure, the copy operation could still have been performed successfully.

The error message is due to the selected data type.

Could an error not arise here? The caller should consider the possibility of the stack being empty.

Realizing Stacks on an Array

- The code we have written correctly realizes the member functions of a stack, but it places a constraint on the maximum number of elements the stack may contain.
- This shortcoming results from using an array to store the data. (Arrays have constant sizes, and their sizes need to be decided in advance.) In fact, this is not the desired stack.
- In this realization, the stack becomes full after accepting a certain number of elements. The push operation returns an error message in the case of the stack being full!

Realizing Stacks Using Lists

- The restrictions imposed as a result of using an array to realize a stack can be removed by using linked lists.
- In list operations, it is faster to add to and remove from the beginning of the list. The whole list has to be traversed to find the end of the list.
- That is why the top of the stack is designed to be first element of the linked list.
- Pushing onto the stack can be interpreted as adding an element to the beginning of the list.
- Popping from the stack can be interpreted as removing an element from the beginning of the list.

Stack Data Structure

```
{#define STACKSIZE 5  
typedef char StackDataType;}
```

```
struct Stack{
```

```
{ StackDataType element[STACKSIZE];  
int top; }
```

```
void create();
```

```
void close();
```

```
bool push(StackDataType);
```

```
StackDataType pop();
```

```
bool isempty();
```

```
};
```

```
typedef char StackDataType;  
struct Node{  
StackDataType data;  
Node *next;  
};
```

```
Node *head;
```

```
void push(StackDataType);
```

```
#include <iostream>
#include <string.h>
#include "stack_1.h"

void Stack::create(){
    head = NULL;
}

void Stack::close(){
    Node *p;
    while (head){
        p = head;
        head = head->next;
        delete p;
    }
}
```

```
void Stack::push(StackDataType newdata){
    Node *newnode = new Node;
    newnode->data = newdata;
    newnode->next = head;
    head = newnode;
}

StackDataType Stack::pop(){
    Node *topnode;
    StackDataType temp;
    topnode = head;
    head = head->next;
    temp = topnode->data;
    delete topnode;
    return temp;
}

bool Stack::isempty(){
    return head == NULL;
}
```

```

int main(){

    Stack s;
    s.create();
    s.push('A');
    s.push('B');
    char c = s.pop();
    c = s.pop();
    // if (!s.isEmpty())
    //     s.pop();      →
    s.close();

    return EXIT_SUCCESS;
}

```

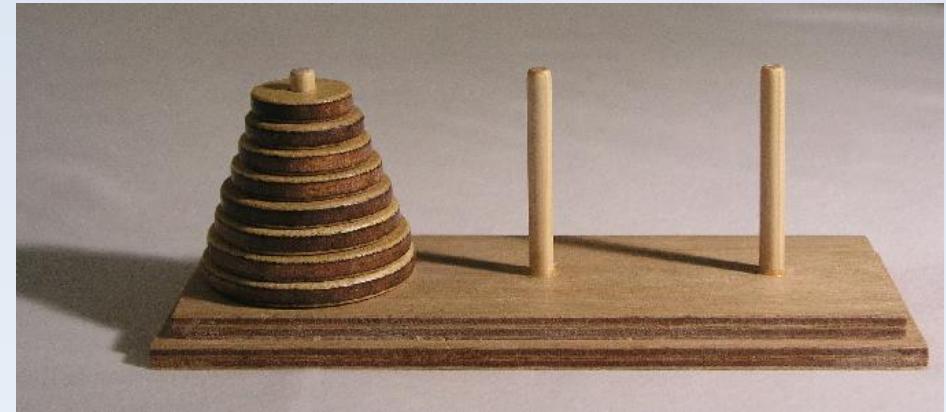
```

H:\mydocuments\dersler\veriyap\yenisular\kodlar\hafta6>a.exe
    3 [main] a 6072 _cygtls::handle_exceptions: Exception: STATUS_ACCESS_VIOLATION
    11292 [main] a 6072 open_stackdumpfile: Dumping stack trace to a.exe.stackdump
    3 [main] a 6072 _cygtls::handle_exceptions: Exception: STATUS_ACCESS_VIOLATION
    11292 [main] a 6072 open_stackdumpfile: Dumping stack trace to a.exe.stackdump
    333205 [main] a 6072 _cygtls::handle_exceptions: Exception: STATUS_ACCESS_VIOLATION
    341659 [main] a 6072 _cygtls::handle_exceptions: Error while dumping state (probably corrupted stack)

```

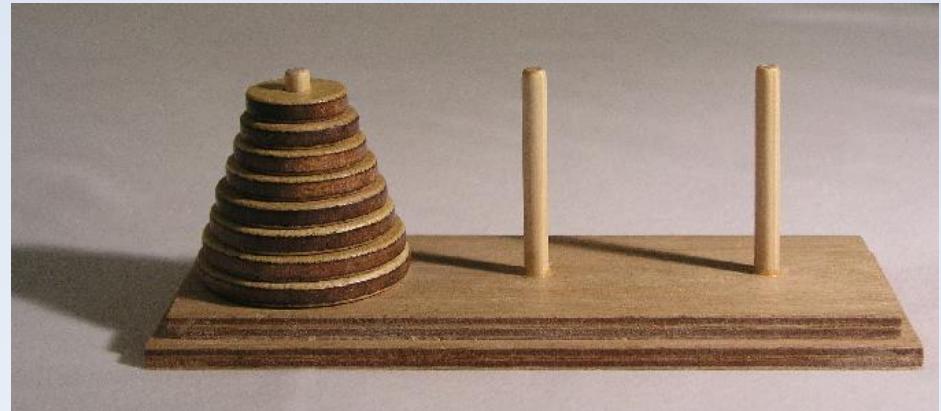
Since the stack is empty, the pop operation will give an error. This error will not show up during the compilation phase, but will occur when the program is running.

Towers of Hanoi



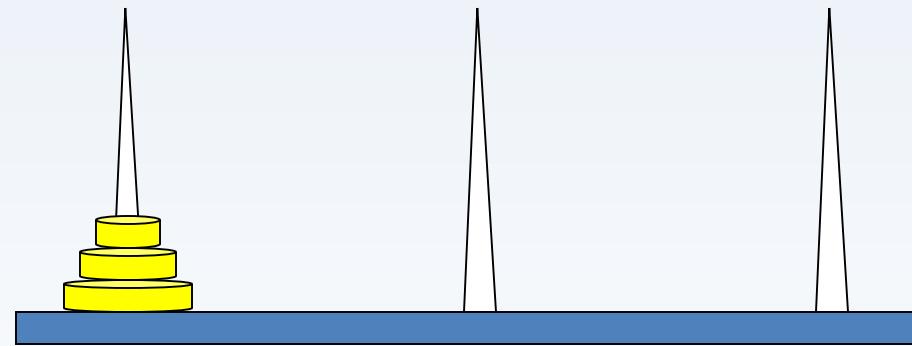
- **Towers of Hanoi** is a mathematical game.
- It consists of three pegs and disks of various sizes.
- You can slide these disks onto any peg you like.
- The puzzle starts off with the disks ordered from smallest to largest on a peg (the smallest disk is at the top).
- Thus, a conical shape has been created.

Towers of Hanoi

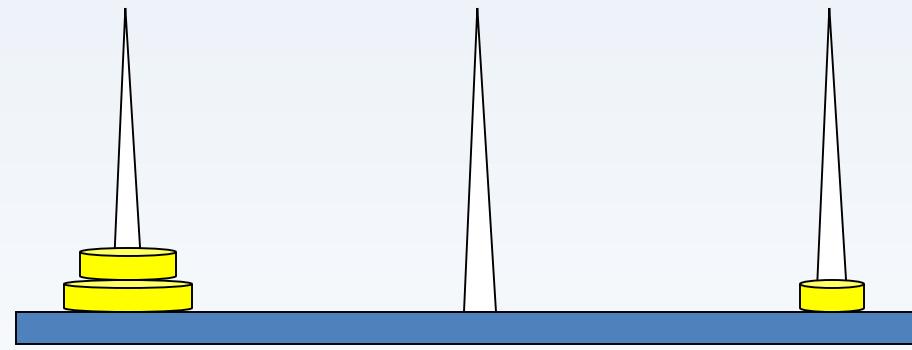


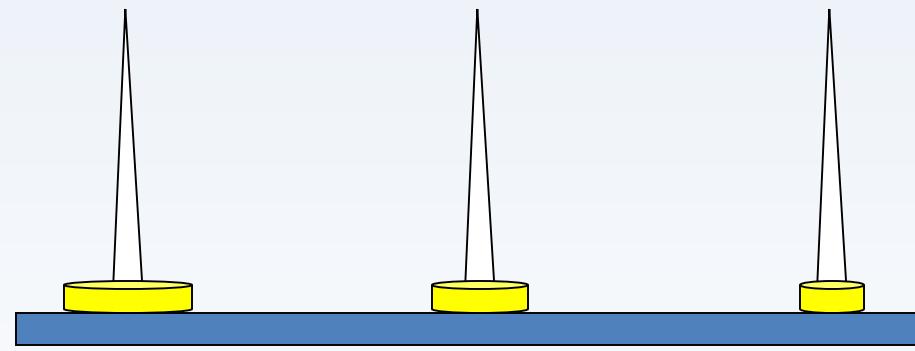
The aim of the game is to move all disks to another peg based on the following rules:

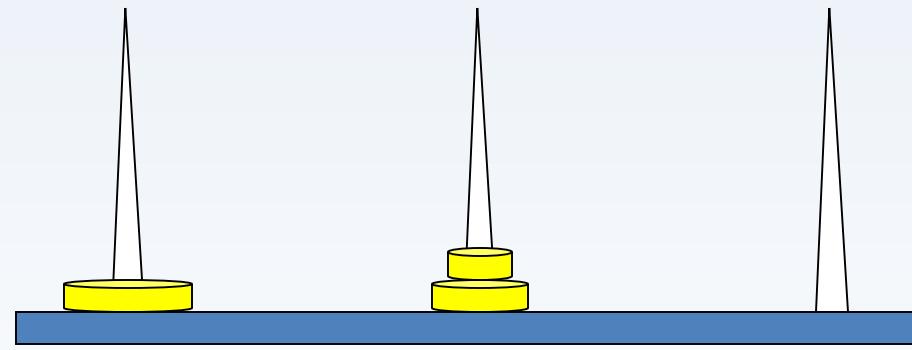
- In each move, we can move only one disk.
- Each move consists of removing the topmost disk from one of the pegs and sliding it onto another peg. Other disks may already be present on this new peg.
- No disk may be placed on top of a disk smaller than itself.

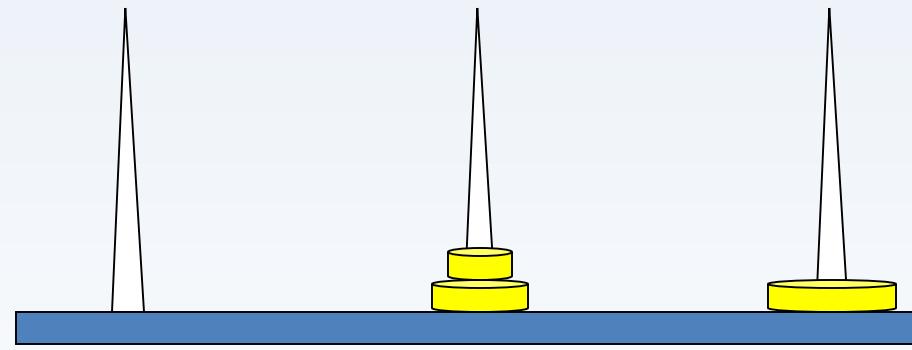


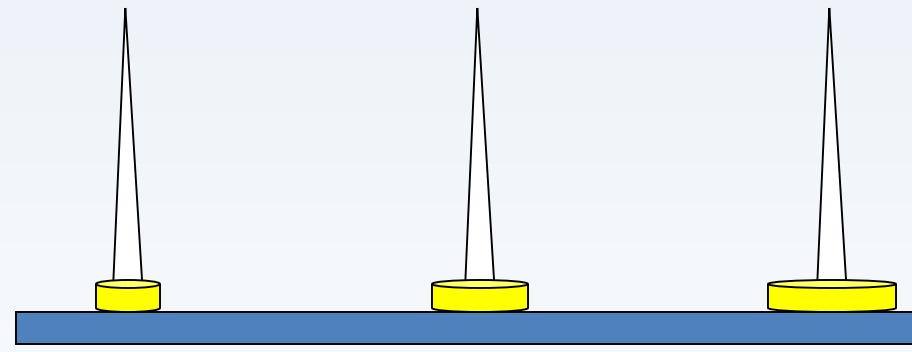
$n = 3$

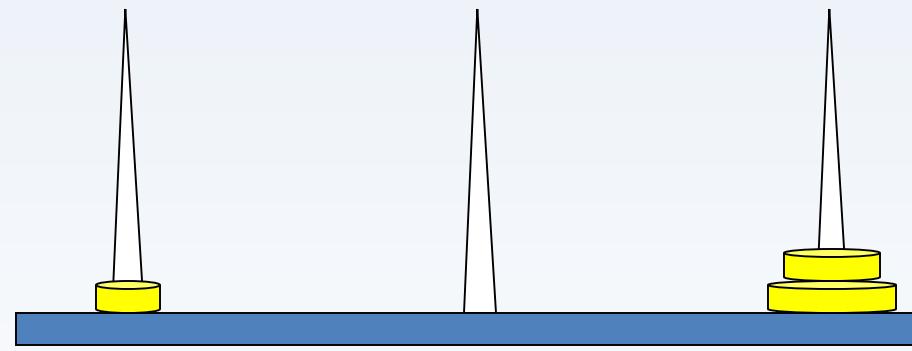


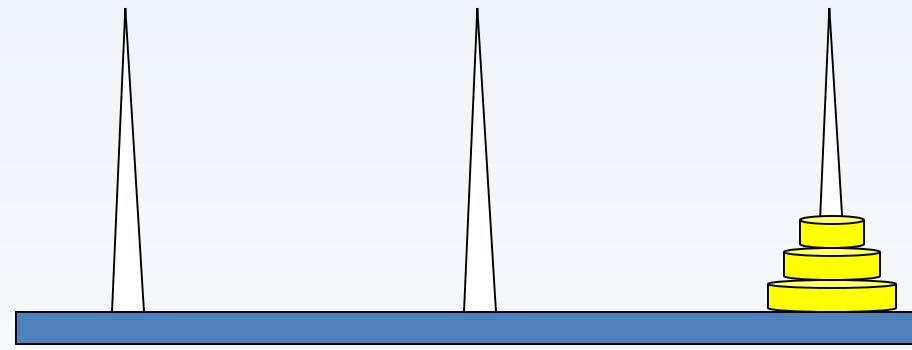


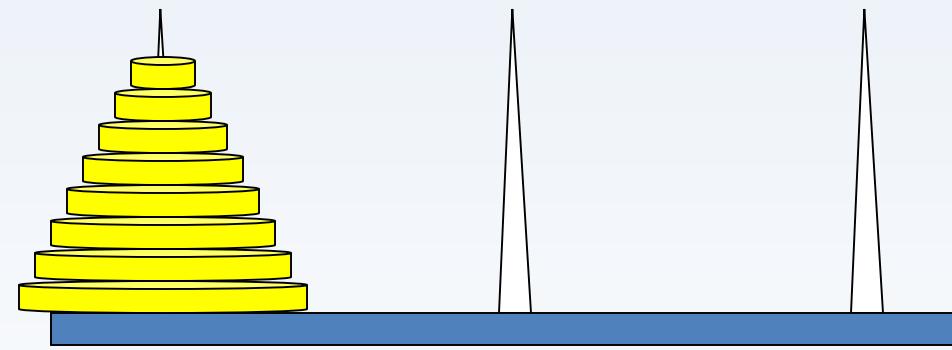




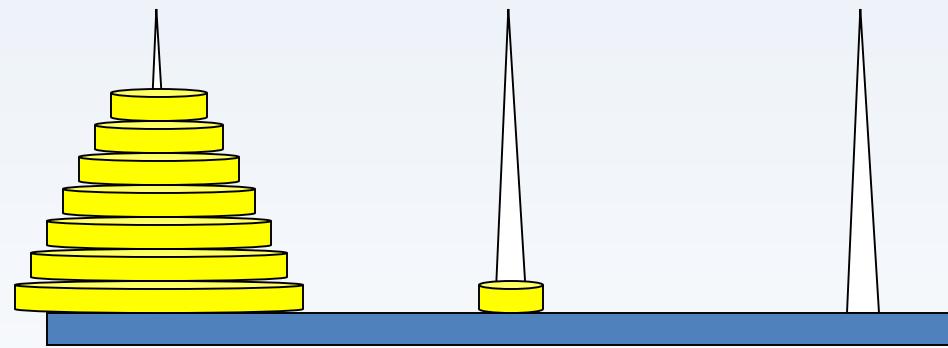




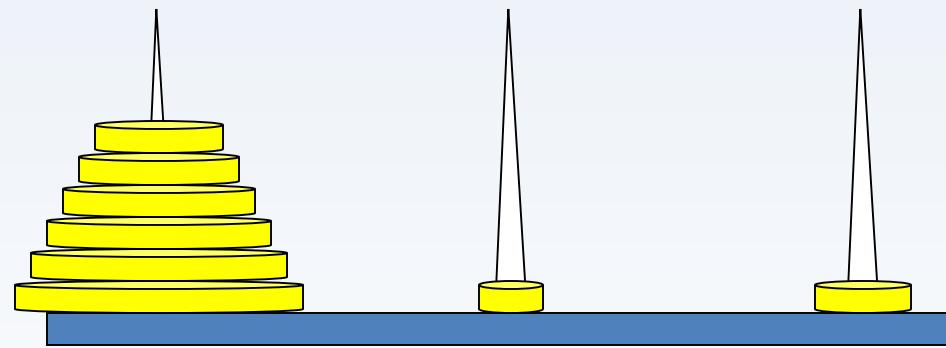




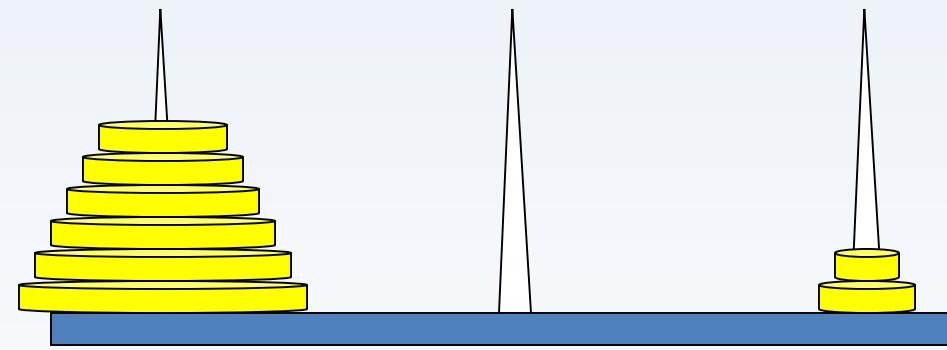
$n = 8$



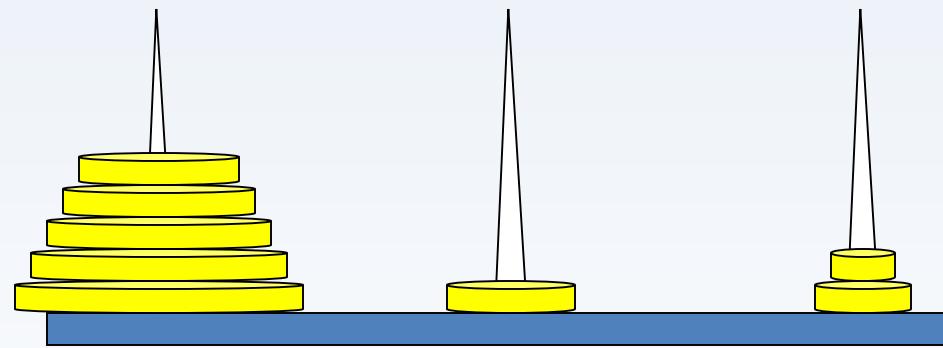
$n = 8$



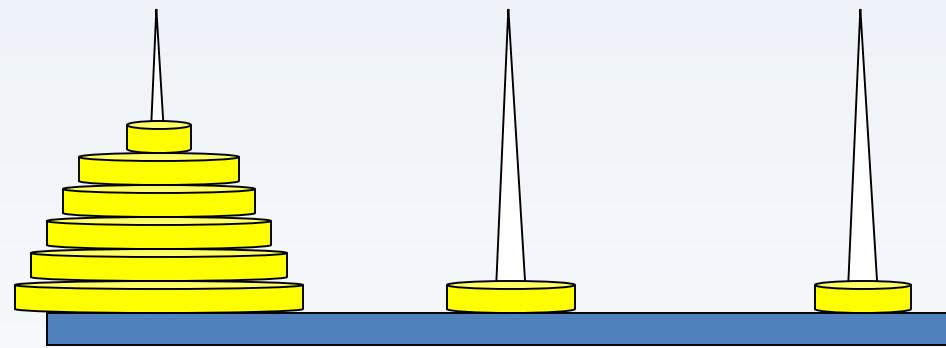
$n = 8$



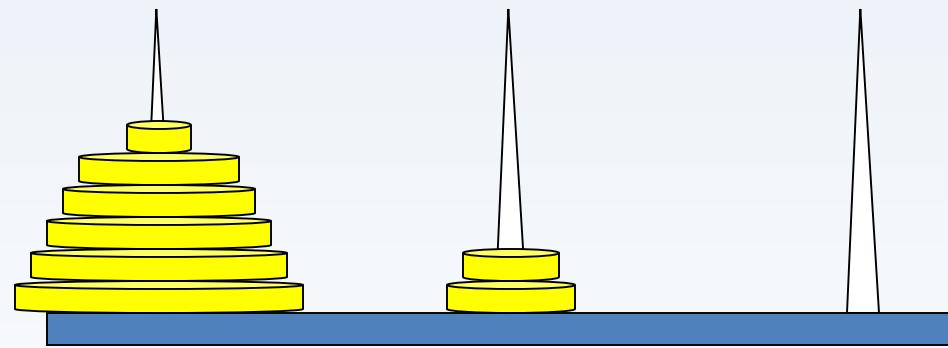
$n = 8$



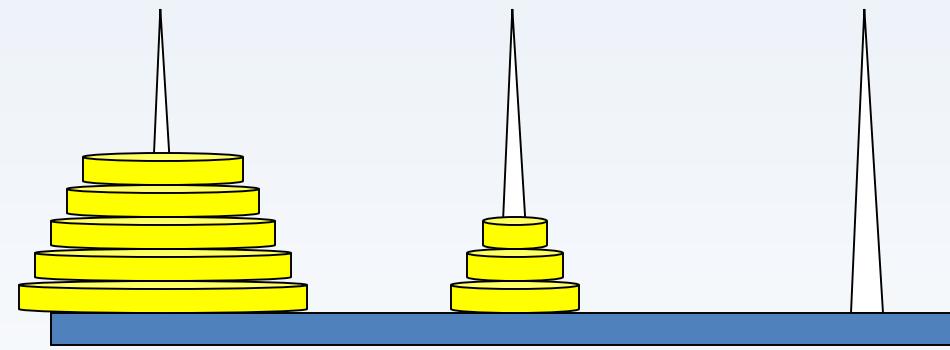
$n = 8$



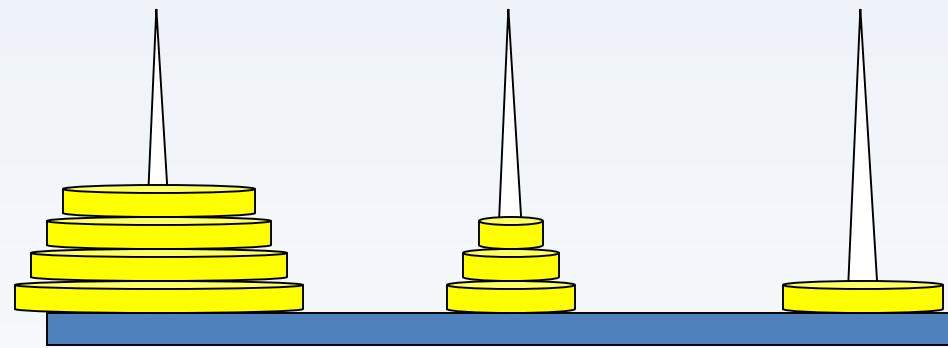
$n = 8$



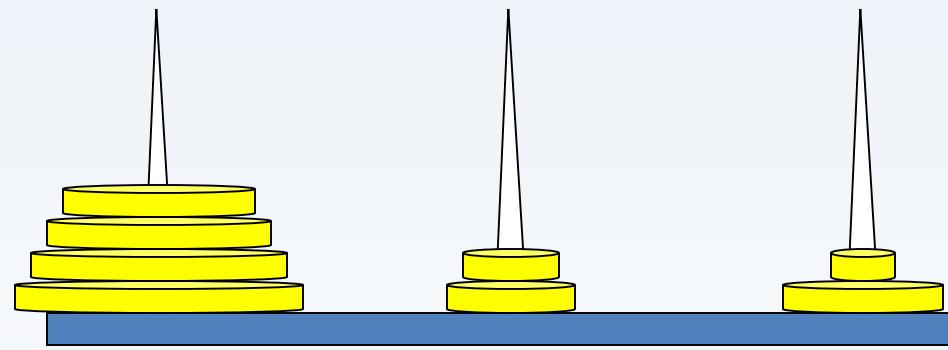
$n = 8$



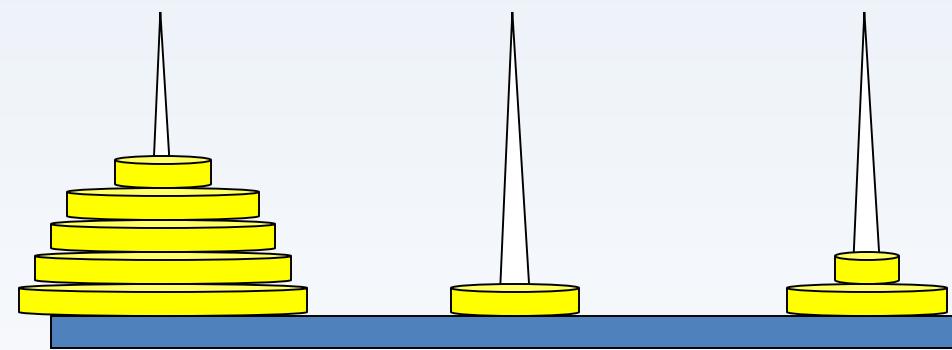
$n = 8$



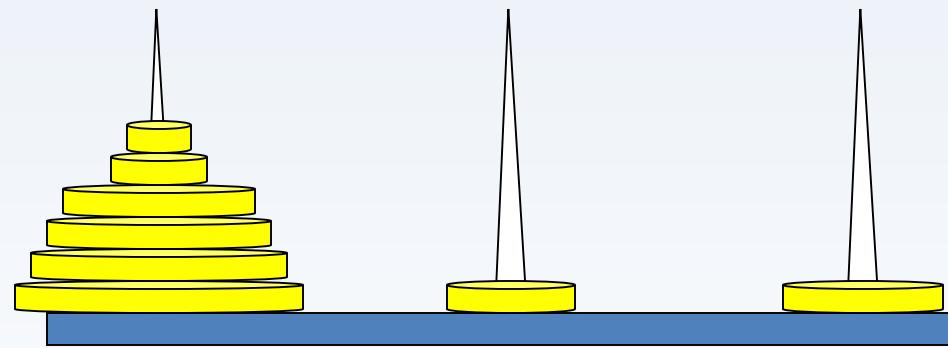
$n = 8$



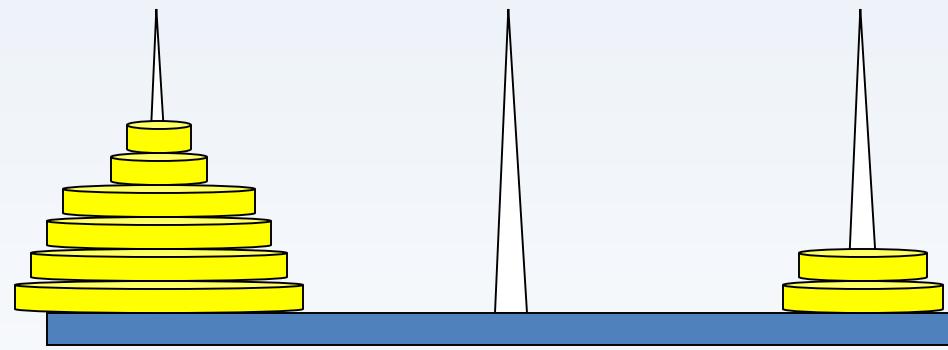
$n = 8$



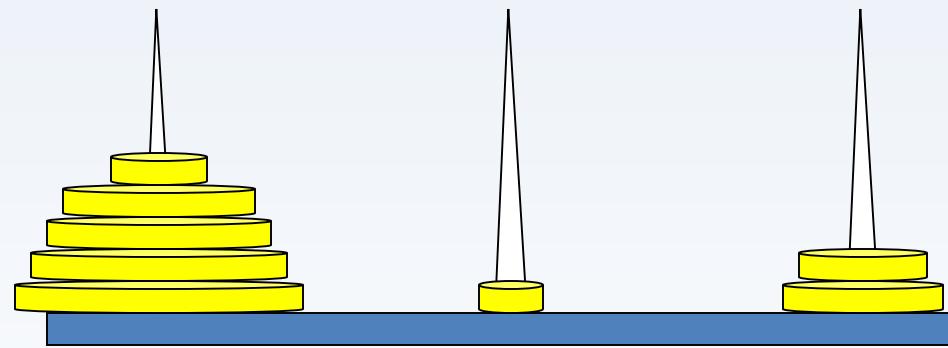
$n = 8$



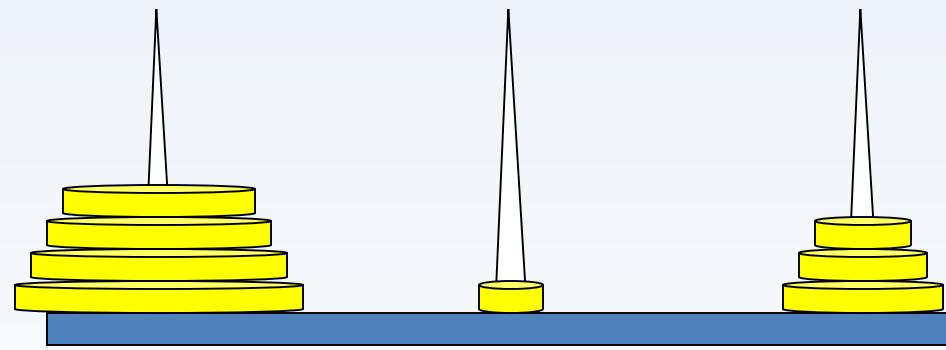
$n = 8$



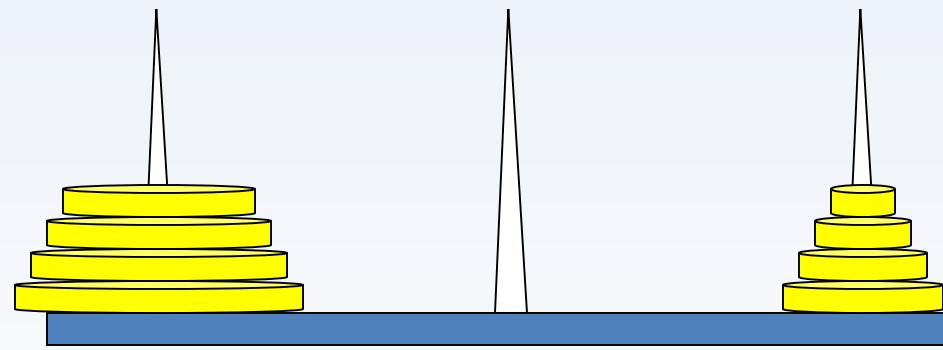
$n = 8$



$n = 8$



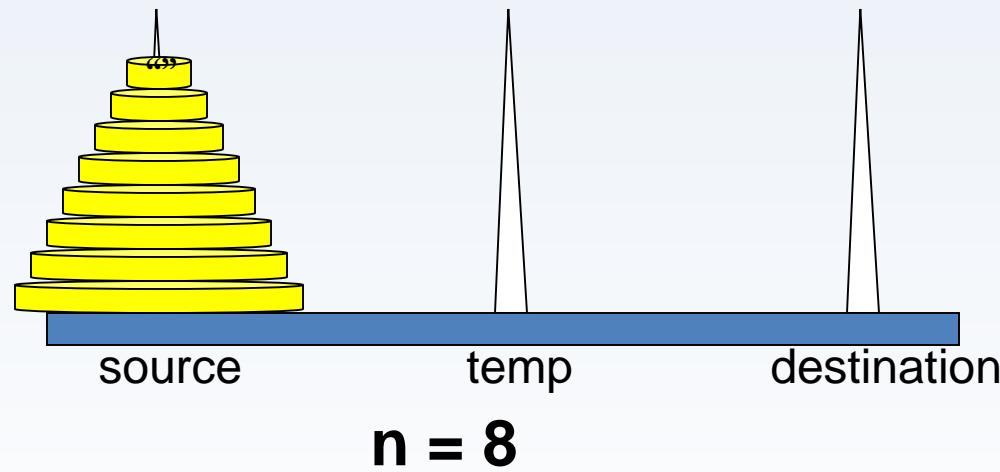
$n = 8$



$n = 8$

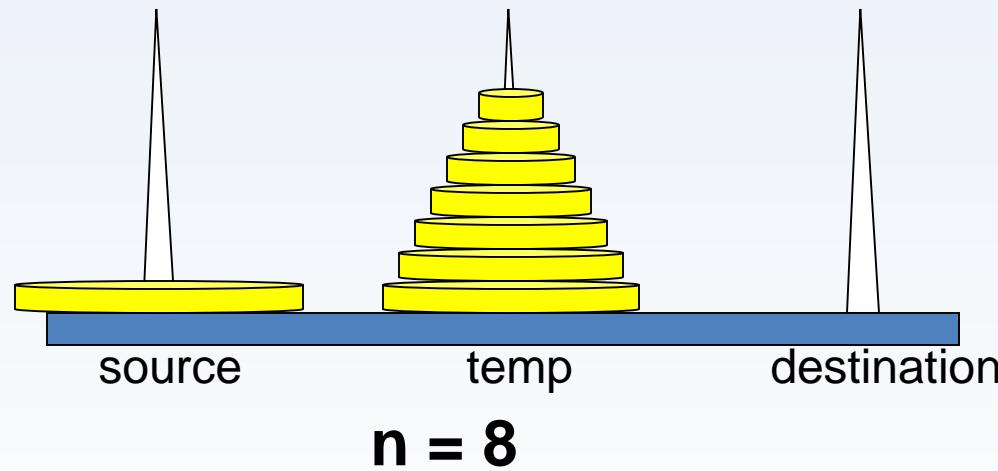
Problem Solution

First, $n - 1$ disks are copied to a temporary place.



Problem Solution

First, $n - 1$ disks are copied to a temporary place.
Then, the n th disk gets copied to the destination.

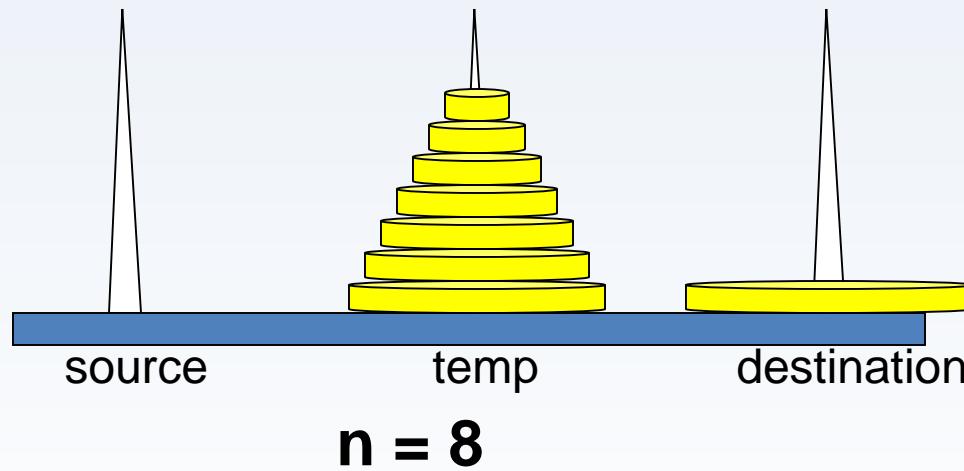


Problem Solution

First, $n - 1$ disks are copied to a temporary place.

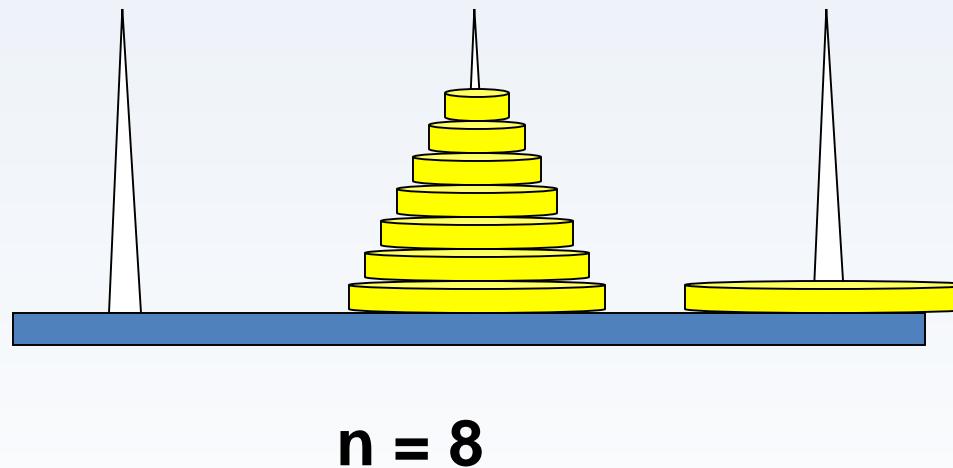
Then, the n th disk gets copied to the destination.

The problem has been solved when $n - 1$ disks in the temporary place have been copied to the destination.



Problem Solution

We try to solve the problem of $n - 1$ disks being copied to a temporary place similarly:
 $n - 2$ disks are copied to a temporary place.
Thus, $(n - 1)$ st disk is copied to a place above n th disk.



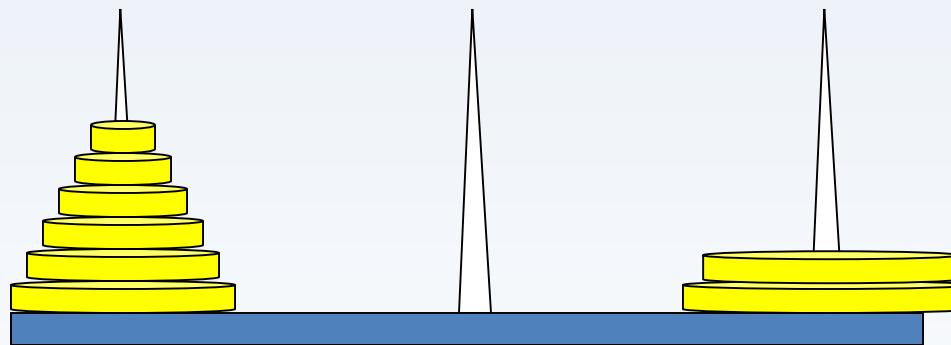
Problem Solution

$n - 2$ disks are copied to a temporary place.

Thus, $(n - 1)$ st disk is copied to a place above n th disk.

We continue the operation in this manner by decreasing n .

The movement of each $(n - x)$ th disk from one place to another should be handled as a similar subproblem.



$$n = 8$$

Problem Solution

First, $n - 1$ disks are copied to a temporary place.

Then, the n th disk gets copied to the destination.

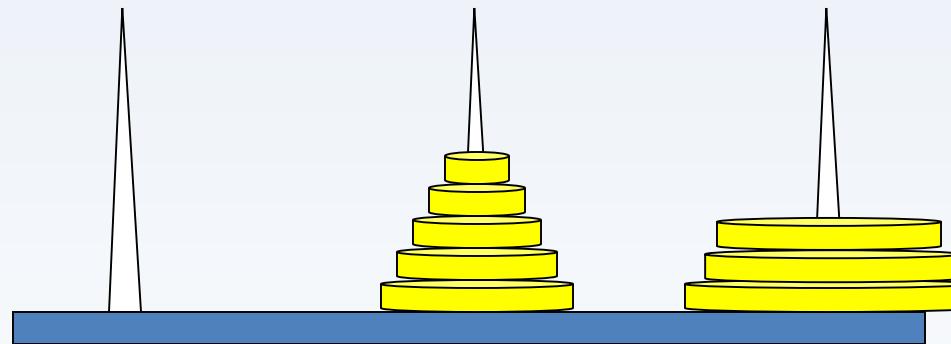
We try to solve the problem of $n - 1$ disks being copied to a temporary place similarly.

Then, $n - 2$ disks are copied to a temporary place.

Thus, $(n-1)$ th disk is copied to a place above n th disk.

We continue the operation in this manner by decreasing n .

The movement of each $(n-x)$ th disk from one place to another should be handled as a similar subproblem.



$$n = 8$$

Problem Solution

First, $n - 1$ disks are copied to a temporary place.

Then, the n th disk gets copied to the destination.

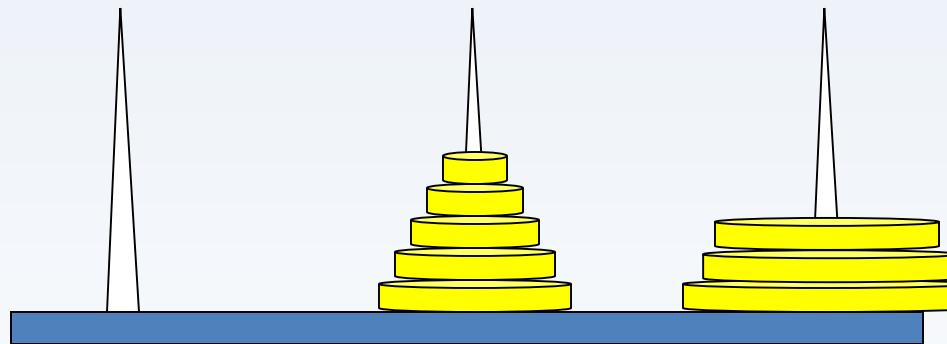
We try to solve the problem of $n - 1$ disks being copied to a temporary place similarly.

Then, $n - 2$ disks are copied to a temporary place.

Thus, $(n-1)$ th disk is copied to a place above n th disk.

We continue the operation in this manner by decreasing n .

The movement of each $(n-x)$ th disk from one place to another should be handled as a similar subproblem.



$$n = 8$$

Problem Solution

First, $n - 1$ disks are copied to a temporary place.

Then, the n th disk gets copied to the destination.

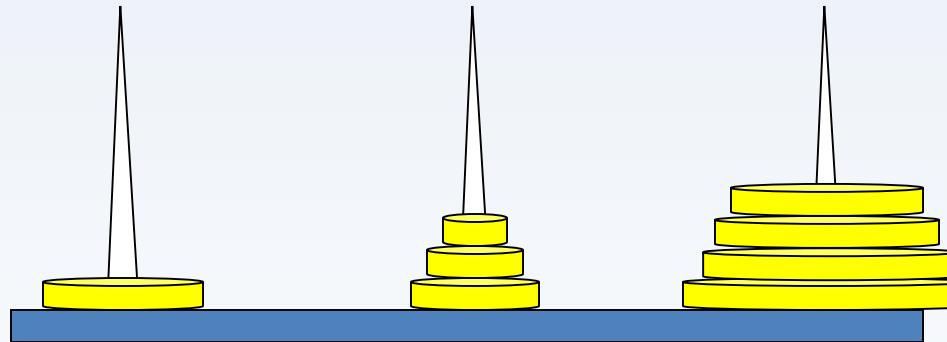
We try to solve the problem of $n - 1$ disks being copied to a temporary place similarly.

Then, $n - 2$ disks are copied to a temporary place.

Thus, $(n-1)$ th disk is copied to a place above n th disk.

We continue the operation in this manner by decreasing n .

The movement of each $(n-x)$ th disk from one place to another should be handled as a similar subproblem.



$$n = 8$$

Problem Solution

First, $n - 1$ disks are copied to a temporary place.

Then, the n th disk gets copied to the destination.

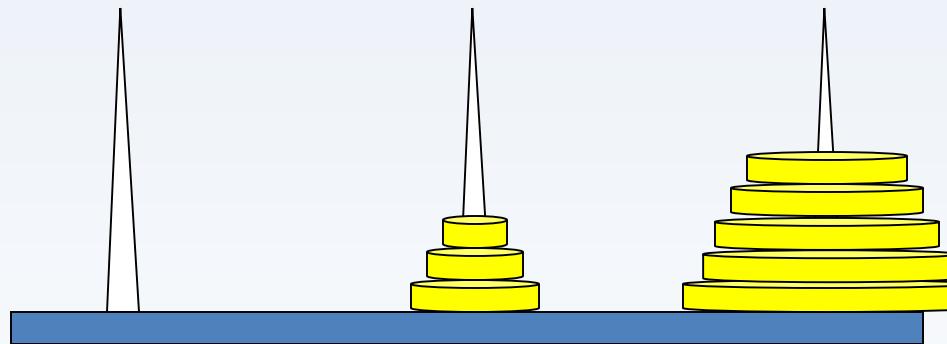
We try to solve the problem of $n - 1$ disks being copied to a temporary place similarly.

Then, $n - 2$ disks are copied to a temporary place.

Thus, $(n-1)$ th disk is copied to a place above n th disk.

We continue the operation in this manner by decreasing n .

The movement of each $(n-x)$ th disk from one place to another should be handled as a similar subproblem.



$$n = 8$$

Problem Solution

First, $n - 1$ disks are copied to a temporary place.

Then, the n th disk gets copied to the destination.

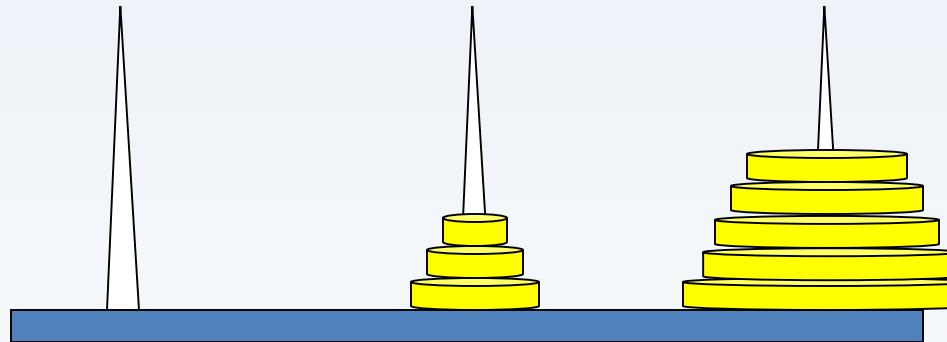
We try to solve the problem of $n - 1$ disks being copied to a temporary place similarly.

Then, $n - 2$ disks are copied to a temporary place.

Thus, $(n-1)$ th disk is copied to a place above n th disk.

We continue the operation in this manner by decreasing n .

The movement of each $(n-x)$ th disk from one place to another should be handled as a similar subproblem.



$$n = 8$$

Problem Solution

First, $n - 1$ disks are copied to a temporary place.

Then, the n th disk gets copied to the destination.

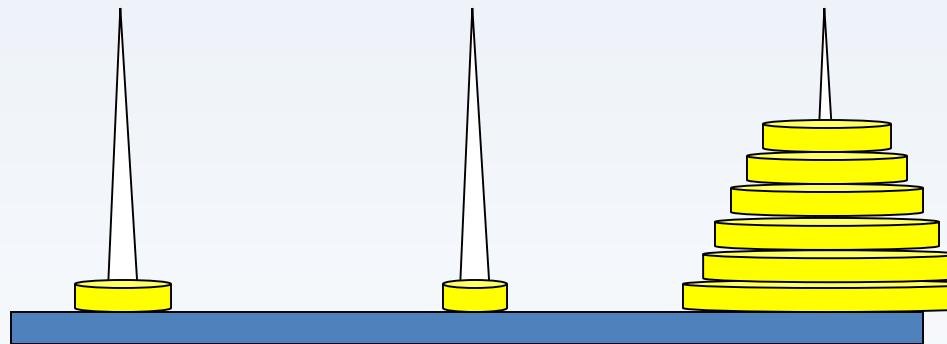
We try to solve the problem of $n - 1$ disks being copied to a temporary place similarly.

Then, $n - 2$ disks are copied to a temporary place.

Thus, $(n-1)$ th disk is copied to a place above n th disk.

We continue the operation in this manner by decreasing n .

The movement of each $(n-x)$ th disk from one place to another should be handled as a similar subproblem.



$$n = 8$$

Problem Solution

First, $n - 1$ disks are copied to a temporary place.

Then, the n th disk gets copied to the destination.

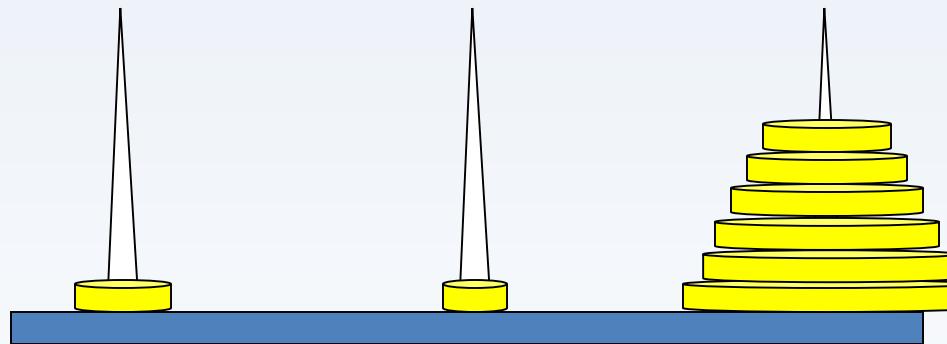
We try to solve the problem of $n - 1$ disks being copied to a temporary place similarly.

Then, $n - 2$ disks are copied to a temporary place.

Thus, $(n-1)$ th disk is copied to a place above n th disk.

We continue the operation in this manner by decreasing n .

The movement of each $(n-x)$ th disk from one place to another should be handled as a similar subproblem.

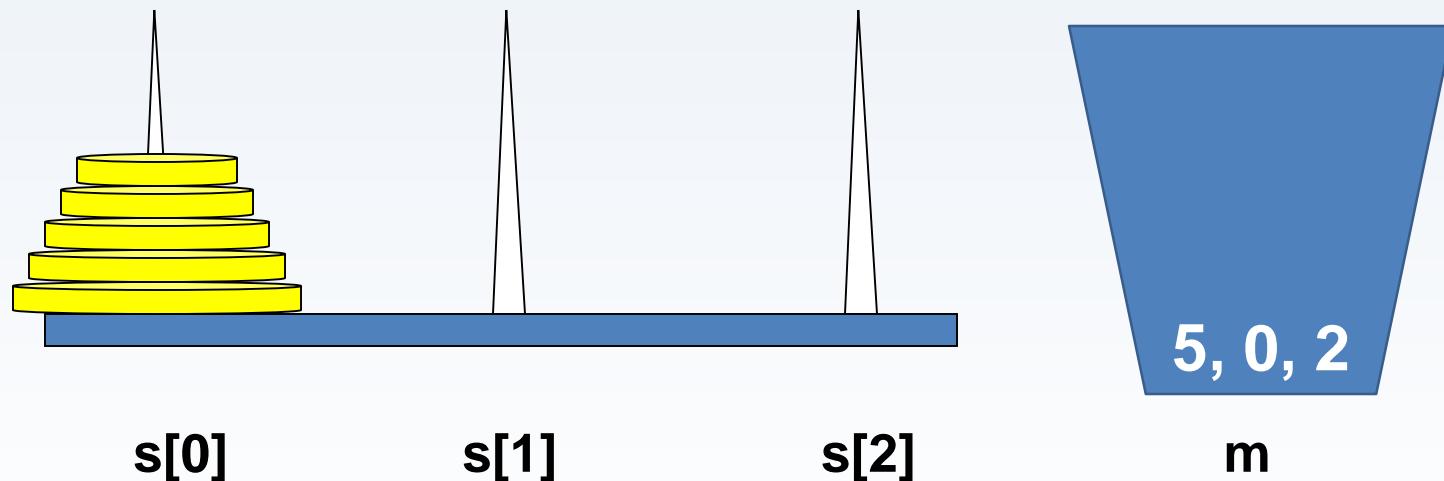


$$n = 8$$

The problem could be solved iteratively using 4 stacks:

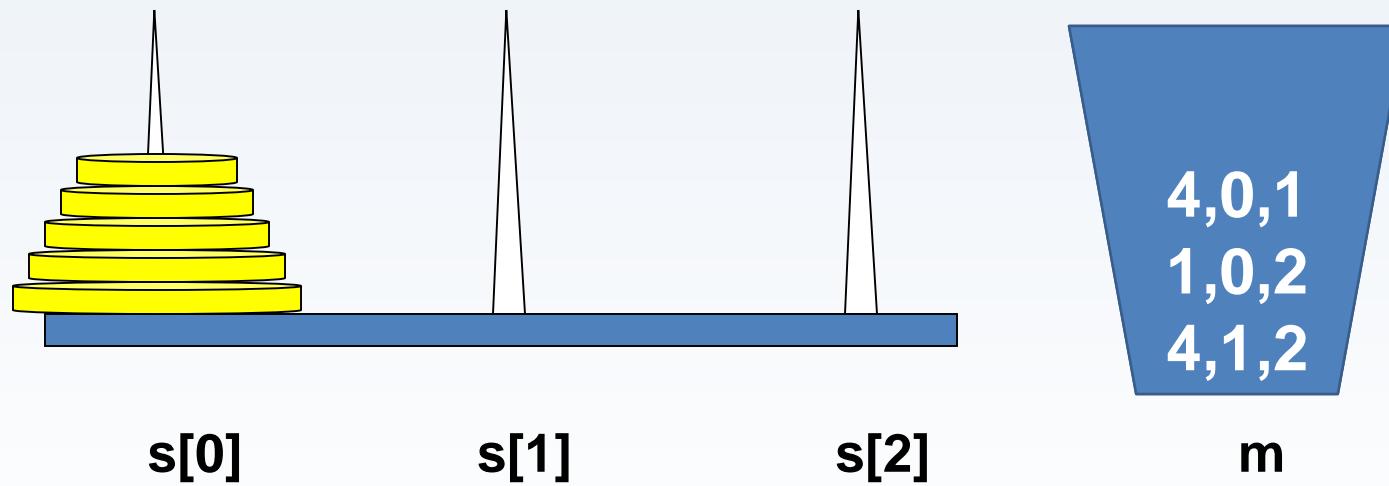
- 3 stacks for the pegs that hold the disks,
- 1 move stack that stores the moves as the operations are being performed

- The program starts by sliding n disks onto the first peg.
- The first move (that is, the goal of the system, i.e., n disks being transferred from stack 1 to stack 3) is copied onto the move stack
 - example: 5, 0, 2: five disks being copied from $s[0]$ to $s[2]$



```
stack s[3]; // the 3 pegs that hold the disks
int main() {
    for (int i = 0; i < 3; i++) {
        s[i].create();
    }
    for (int i = 0; i < 5; i++) { // disk n = 5
        s[0].push(5 - i); // pushed onto stack 0
    }
    Hanoi_iterative(5);
    for (int i = 0; i < 3; i++) {
        s[i].close();
    }
    return EXIT_SUCCESS;
}
```


- Then, as long as the move stack is full, we pop a move (n , source, destination) from this stack, and push these moves to the stack:
 - ($n-1$, temp, destination)
 - (1, source, destination)
 - ($n-1$, source, temp) (the move that should be executed first is at the top of the stack)



```

void Hanoi_iterative(int n) {
    Move_Stack m;
    m.create();
    StackMoveType move = {n, 0, 2};
    m.push(move);
    while ( !m.isempty() ) {
        move = m.pop();
        if (move.n == 1)
            s[move.destination].push(s[move.source].pop());
        else {
            int temp = 0 + 1 + 2 - move.destination - move.source;
            StackMoveType newmove = {move.n-1, temp, move.destination};
            m.push(newmove);
            newmove.n = 1;
            newmove.source = move.source;
            newmove.destination = move.destination;
            m.push(newmove);
            newmove.n = move.n-1;
            newmove.source = move.source;
            newmove.destination = temp;
            m.push(newmove);
        }
    }
    m.close();
}

```

To do: In recitation

Infix - Postfix Conversion

Data Structures

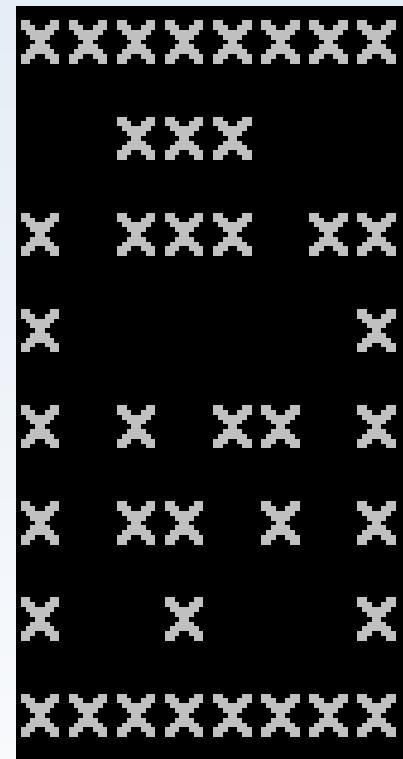
Stack Applications

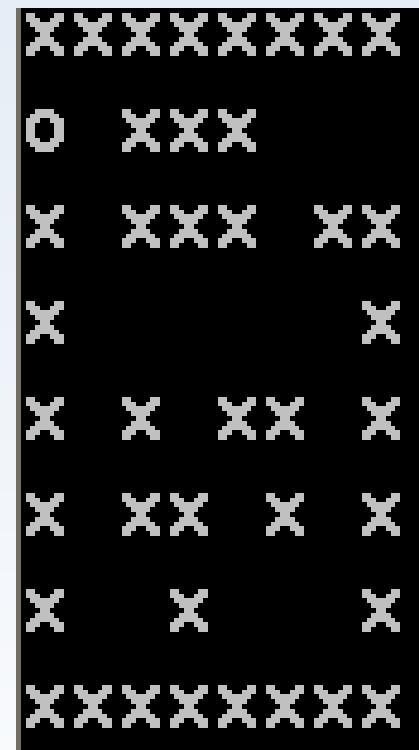
Finding a Path in a Labyrinth

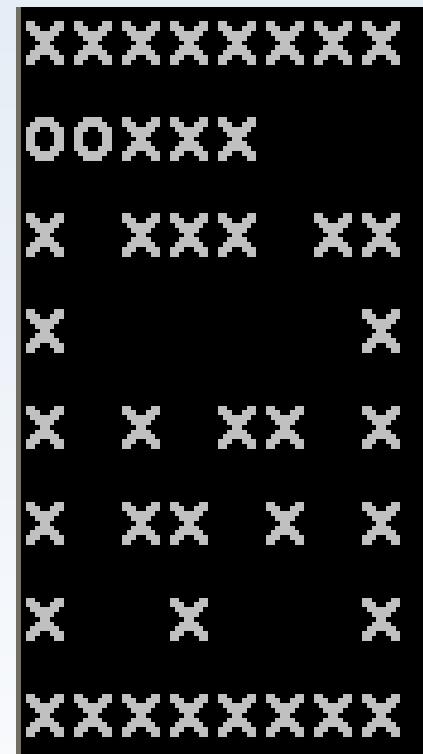
- The problem of finding a path in a labyrinth can be solved using the stack data structure:
 - While traversing the labyrinth, the states at decision points where going in more than one direction is possible get pushed onto a stack.
- We select one of the possible directions and proceed in that direction.
- If the choice made is not a correct choice, and we cannot find the exit of the labyrinth in this way, we go back to the last decision point (by popping the last state from the stack) and continue to search for the exit in the other untraversed directions.
- In the example labyrinth below, the first four steps are shown.
- In this representation, x's represent walls, the empty spaces represent paths, and o's represent traversed positions.

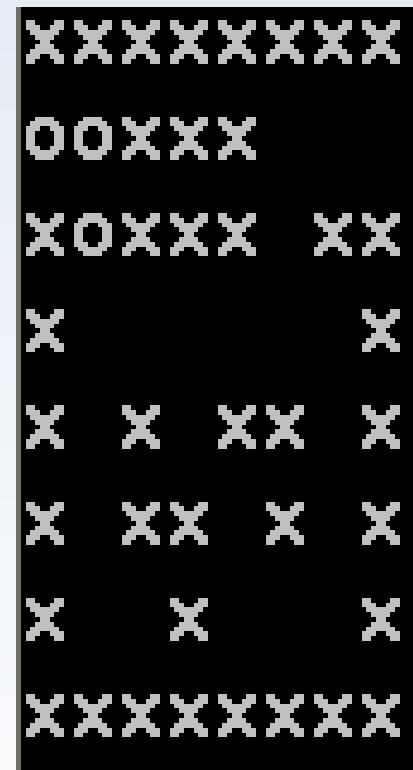
ENTRANCE →

→ EXIT









DECISION
POINT(D1) 



STACK

XXXXXX	XXXXXX	XXXXXX	XXXXXX
OOXX	OOXX	OOXX	OOXX
XOXX XX	XOXX XX	XOXX XX	XOXX XX
XO X	XO X	XO X	XO X
XOX XX X	XOX XX X	XOX XX X	XOX XX X
X XX X X	XOXX X X	XOXX X X	XOXX X X
X X X X	X X X X	X O X X	XOOX X
XXXXXX	XXXXXX	XXXXXX	XXXXXX

XXXXXX	XXXXXX	XXXXXX	XXXXXX
OOXX	OOXX	OOXX	OOXX
XOXX XX	XOXX XX	XOXX XX	XOXX XX
XO X	XO X	XO X	XO X
XOX XX X	XOX XX X	XOX XX X	XOX XX X
X XX X X	XOXX X X	XOXX X X	XOXX X X
X X X X	X X X X	X O X X	XOOX X
XXXXXX	XXXXXX	XXXXXX	XXXXXX

XXXXXX	XXXXXX	XXXXXX	XXXXXX
OOXX	OOXX	OOXX	OOXX
XOXX XX	XOXX XX	XOXX XX	XOXX XX
XO X	XO X	XO X	XO X
XOX XX X	XOX XX X	XOX XX X	XOX XX X
X XX X X	XOXX X X	XOXX X X	XOXX X X
X X X X	X X X X	X O X X	XOOX X
XXXXXX	XXXXXX	XXXXXX	XXXXXX

XXXXXX	XXXXXX	XXXXXX	XXXXXX
OOXX	OOXX	OOXX	OOXX
XOXX XX	XOXX XX	XOXX XX	XOXX XX
XO X	XO X	XO X	XO X
XOX XX X	XOX XX X	XOX XX X	XOX XX X
X XX X X	XOXX X X	XOXX X X	XOXX X X
X X X X	X X X X	X O X X	XOOX X
XXXXXX	XXXXXX	XXXXXX	XXXXXX



CANNOT MOVE!
GO BACK TO LAST
DECISION POINT

XXXXXX	XXXXXX	XXXXXX
OOXXX	OOXXX	OOXXX
XOXXX XX	XOXXX XX	XOXXX XX
XOO X	XOOO X	XOOO X
XOX XX X	XOX XX X	XOXOXX X
XOXX X X	XOXX X X	XOX X X
XOOX X	XOOX X	XOOX X
XXXXXX	XXXXXX	XXXXXX

XXXXXX	XXXXXX	XXXXXX
OOXXX	OOXXX	OOXXX
XOXXX XX	XOXXX XX	XOXXX XX
XOO X	XOOO X	XOOO X
XOX XX X	XOX XX X	XOXOXX X
XOXX X X	XOXX X X	XOX X X
XOOX X	XOOX X	XOOX X
XXXXXX	XXXXXX	XXXXXX

XXXXXX	XXXXXX	XXXXXX
OOXXX	OOXXX	OOXXX
XOXXX XX	XOXXX XX	XOXXX XX
XOO X	XOOO X	XOOO X
XOX XX X	XOX XX X	XOXOXX X
XOXX X X	XOXX X X	XOX X X
XOOX X	XOOX X	XOOX X
XXXXXX	XXXXXX	XXXXXX

```
typedef struct d{  
    int x;  
    int y;  
    int right;  
    int left;  
    int down;  
    int up;  
    int camefrom;  
}StackDataType, position;
```

```
struct Node{  
    StackDataType data;  
    Node *next;  
};
```

Labyrinth.cpp

```
#define RIGHT 1
#define LEFT 2
#define UP 3
#define DOWN 4

char lab[8][8]={{{'x','x','x','x','x','x','x','x'},{
{' ',' ',' ','x','x','x',' ',' ',' '},{
{'x',' ',' ','x','x','x',' ','x','x'},{
{'x',' ',' ',' ',' ',' ',' ',' ','x'},{
{'x',' ',' ','x',' ','x','x',' ','x'},{
{'x',' ',' ','x',' ','x','x',' ','x'},{
{'x',' ',' ',' ','x',' ',' ',' ','x'},{
{'x','x','x','x','x','x','x','x'}}};
```

```
void printlab(char l[8][8]) {  
    for (int i = 0; i < 8; i++) {  
        for (int j = 0; j < 8; j++)  
            cout << l[i][j];  
        cout << endl;  
    }  
    cout << endl << endl;  
}
```

```
int main(){
    stack s;
    s.create();
    position entrance = {0,1,0,0,0,0,0,0};
    position exit = {7,1,0,0,0,0,0,0};
    position p = entrance;
    p.camefrom = LEFT;
    printlab(lab);
    bool goback = false;
```

```
while (p.x != exit.x || p.y != exit.y) {  
    lab[p.y][p.x]='o';  
    printlab(lab);  
    //first find in how many directions we can move  
    if (!goback) { //if not calculated before  
        p.right = 0; p.left = 0; p.down = 0; p.up = 0;  
        if (p.x<7 && lab[p.y][p.x+1]!='x') p.right=1;//right  
        if (p.x>0 && lab[p.y][p.x-1]!='x') p.left=1;//left  
        if (p.y<7 && lab[p.y+1][p.x]!='x') p.down=1;//down  
        if (p.y>0 && lab[p.y-1][p.x]!='x') p.up=1;//up  
    }  
    else goback = false;
```

//here, one of the possible moves is selected

```
bool moved = true;
position past = p;
if (p.down && p.camefrom != DOWN)
    {p.y++; p.camefrom = UP; past.down = 0;}
else if (p.up && p.camefrom != UP)
    {p.y--; p.camefrom = DOWN; past.up = 0;}
else if (p.left && p.camefrom != LEFT)
    {p.x--; p.camefrom = RIGHT; past.left = 0;}
else if (p.right && p.camefrom != RIGHT)
    {p.x++; p.camefrom = LEFT; past.right = 0;}
else moved = false; //one direction (the minimum) is open, but this is the direction we came from
```

```
if (p.x != exit.x || p.y != exit.y) {  
if ( (past.down + past.up + past.right + past.left)>1) {  
    //there is more than one choice, push onto stack and  
    //continue in that chosen direction. Let the choices  
    //you have not selected remain marked on the stack.  
    s.push(past);  
}  
if (!moved) { // has to go back  
    if ( !s.isEmpty() ) {  
        p = s.pop();  
        goback = true;  
    }  
}  
}  
}
```

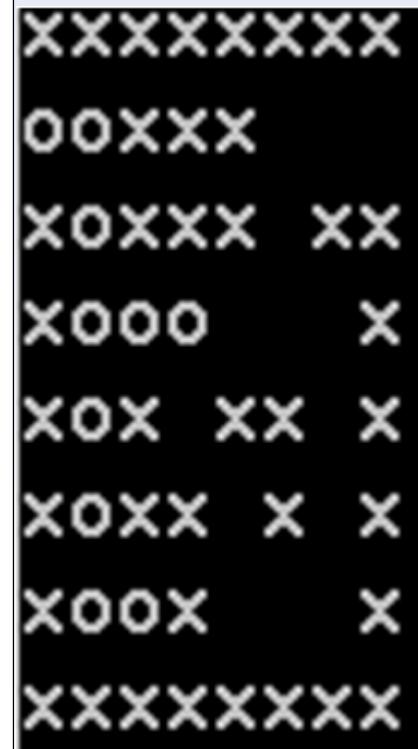
```
 } //end of while
lab[p.y][p.x] = 'o';
printLab(lab);
cout << "PATH found" << endl;
s.close();

return EXIT_SUCCESS;
}
```

- Assuming that the code works step-by-step, the state of the stack after every stack operation (push and pull) is shown below.



s	{head=0x003b6138 size=-858993459 }
head	0x003b6138 {data={...} next=0x00000000 }
data	{x=1 y=3 right=1 ...}
x	1
y	3
right	1
left	0
down	0
up	1
camefrom	3



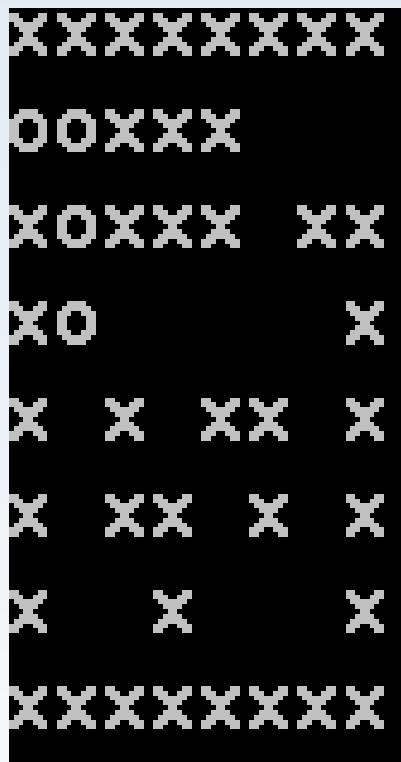
[-] s		{head=0x003b6138 size=-858993459 }
[-] head		0x003b6138 {data={...} next=0x00000000 }
[-] data		{x=3 y=3 right=1 ...}
x		3
y		3
right		1
left		1
down		0
up		0
camefrom		2
[-] next		0x00000000 {data={...} next=??? }



s	{head=0x003b6138 size=-858993459 }
head	0x003b6138 {data={...} next=0x00000000 }
data	{x=5 y=3 right=1 ...}
x	5
y	3
right	1
left	1
down	0
up	0
camefrom	2

- In the part of the code where a direction is selected, as a result of preference being given to going right, the state of the stack after every stack operation (push and pull):

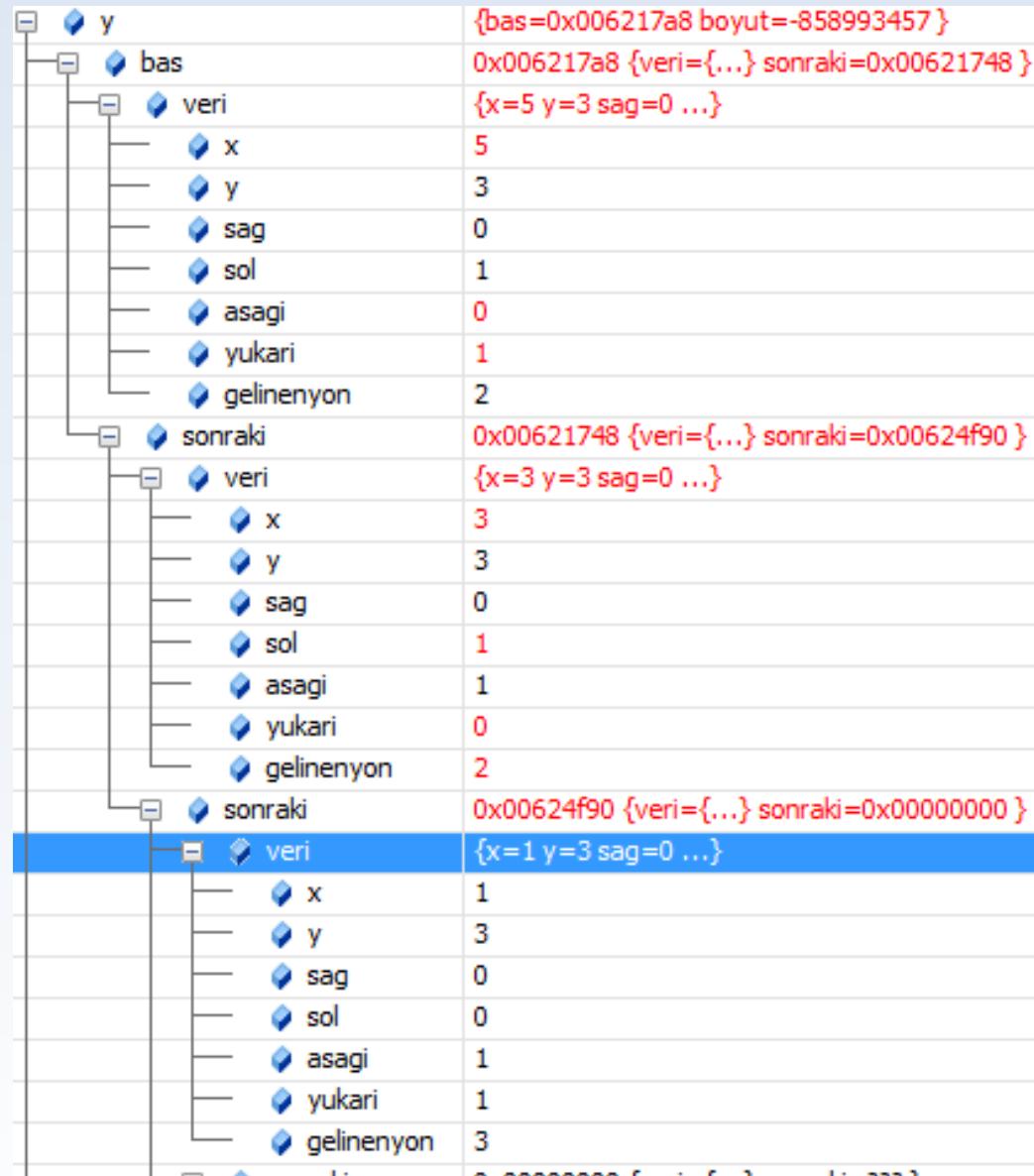
```
if(p.right && p.camefrom != RIGHT)
    {p.x++;p.camefrom=LEFT;past.right=0;}
else if(p.down && p.camefrom != DOWN)
    {p.y++;p.camefrom=UP;past.down=0;}
else if(p.up && p.camefrom != UP)
    {p.y--;p.camefrom=DOWN;past.up=0;}
else if (p.left && p.camefrom != LEFT)
    {p.x--;p.camefrom=RIGHT;past.left=0;}
else moved = false;
```



■	⌚ y	{bas=0x00624f90 boyut=-858993459 }
	└─ bas	0x00624f90 {veri={...} sonraki=0x00000000 }
	└─ veri	{x=1 y=3 sag=0 ...}
	⌚ x	1
	⌚ y	3
	⌚ sag	0
	⌚ sol	0
	⌚ asagi	1
	⌚ yukari	1
	⌚ gelinenyon	3



└── y	{bas=0x00621748 boyut=-858993458 }
└── bas	0x00621748 {veri={...} sonraki=0x00624f90 }
└── veri	{x=3 y=3 sag=0 ...}
└── x	3
└── y	3
└── sag	0
└── sol	1
└── asagi	1
└── yukari	0
└── gelinenyon	2
└── sonraki	0x00624f90 {veri={...} sonraki=0x00000000 }
└── veri	{x=1 y=3 sag=0 ...}
└── x	1
└── y	3
└── sag	0
└── sol	0
└── asagi	1
└── yukari	1
└── gelinenyon	3
└── sonraki	0x00000000 {veri={...} sonraki=??? }



Data Structures

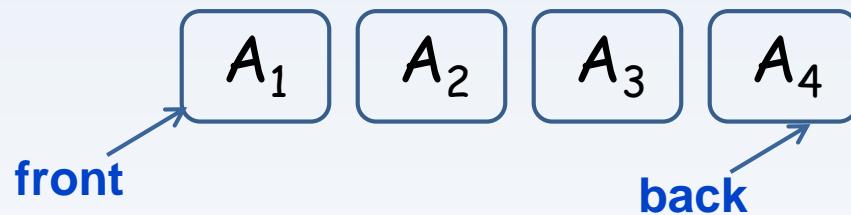
Queue

Queue Structure

- The queue structure is a simple structure like the stack.
- The difference from the stack is that the first element to enter the queue gets processed first.
- That is, service requests get added to the end of the list. That is why, it is also defined as **First In First Out (FIFO)** storage.
- The queue is an ordered waiting list.
- Those requesting a specific service enter an ordered list to get that service.
- The first element to get added to the list gets served first (**First-Come First-Served "FCFS"**).

Queue Structure (continued)

- It is possible to access the structure from two points:
 - a back pointer (back) that marks the position where new arrivals will be added
 - a front pointer (front) to designate the element that will get served



Queue elements get removed only from the front and new elements are added to the back.

Areas of Application

- In computer systems, queues have many areas of application, including:
 - Single-processor computers can only serve one user at a time. User requests are kept in a queue structure.
 - In operating systems, many timing operations are performed using the queue structure.
 - Printing to a printer uses a queue.

Queue Operations

- **enqueue**: operation of adding a new element to the very end of the queue.
 - The added element becomes the last element in the queue.
enqueue(...)
- **dequeue**: operation of removing the foremost element from the queue.
 - The element waiting behind this element gets into the foremost place in the queue.
dequeue()
- **checking for emptiness**: operation of checking if the queue is empty.
isempty()

Exercise

Perform these operations on a queue in order:

- enqueue('A');
- enqueue('B');
- enqueue('C');
- dequeue();
- enqueue('D');
- dequeue();
- dequeue();



Realizing Queues on Arrays

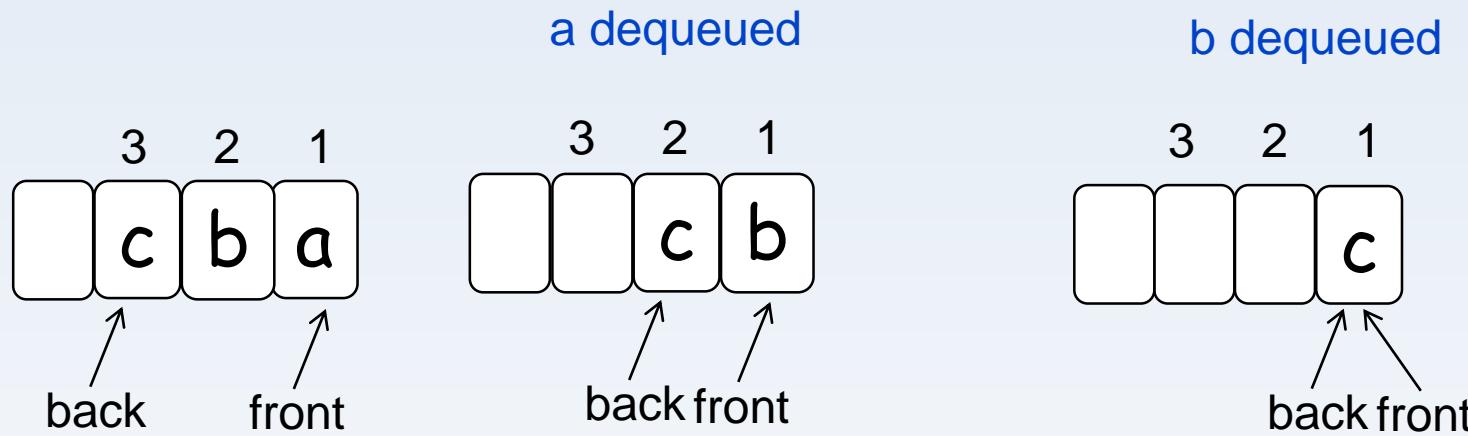
- It is possible to implement a queue on an array.
- We will first look at two implementations, which have some drawbacks.
- Then, we will discuss the proper way.

Realization on an Array 1: Shifting Elements

- Let the queue “front” pointer always be the first element of the array.
- To add a new element, the “back” pointer is incremented once.
- When removing an element, it is always the first element that gets removed.
- But, this is an expensive solution: when removing each element, all other elements have to be shifted once.

Example:

Drawback: Elements must be shifted



Each time, all elements of the array have to be shifted forward once.

Realization on an Array 2: Linear Array

- It is possible to configure front and back pointers in different ways. For example:
 - “back” points to the empty space where the element will be added. To add an element, add (write) and increment the “back” pointer.
 - To remove an element, read the element by using the “front” pointer and then increment the pointer.
 - If we had an array of infinite size, the problem could have been solved as follows:

Not possible!

```
int front = 0; int back = 0;  
char queue[∞]; //infinite!
```

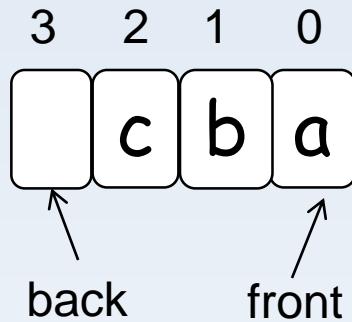
```
bool isempty(){  
    return (front == back);  
}
```

```
void enqueue(char x){  
    queue[back++] = x ;  
}  
  
char dequeue(){  
    return (queue[front++]);  
}
```

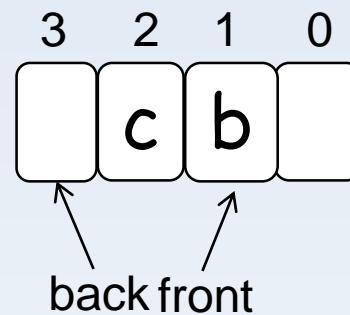
Example:

Drawback: It wastes memory.

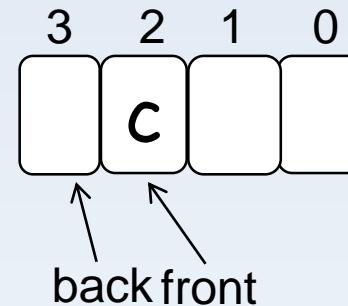
a, b, c enqueued



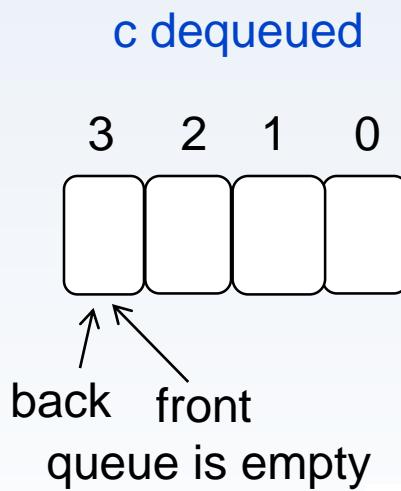
a dequeued



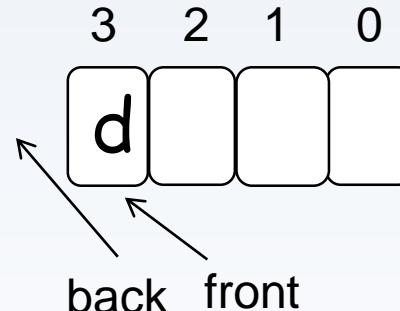
b dequeued



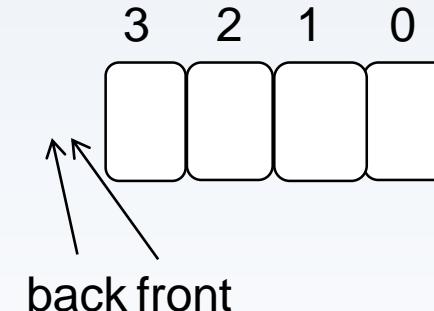
c dequeued



d enqueue



d dequeued



If we have only 4 memory cells for the queue,
no new elements can be enqueue!

It looks like there are no empty spaces in the array.

```
#ifndef QUEUE_H
#define QUEUE_H

#define QUEUESIZE 10

typedef int QueueDataType;
struct Queue {
    QueueDataType element[QUEUESIZE];
    int front;
    int back;
    void create();
    void close();
    bool enqueue(QueueDataType);
    QueueDataType dequeue();
    bool isempty();
};

#endif
```

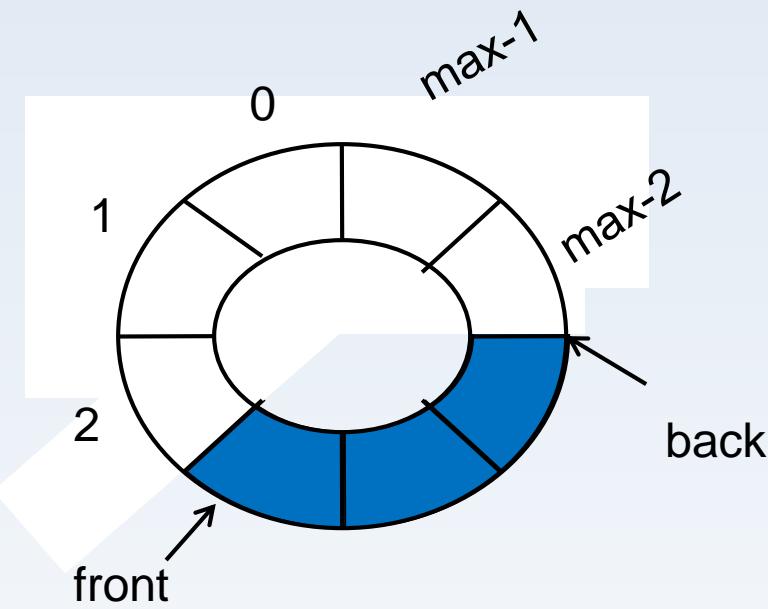
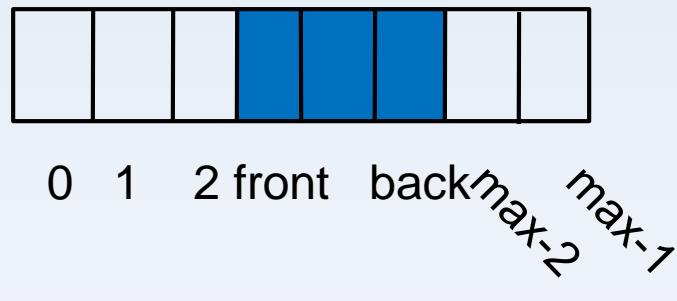
```
void Queue::create() {
    front = 0; back = 0;
}
void Queue::close() {
}
bool Queue::enqueue(QueueDataType newdata) {
    if (back < QUEUESIZE) {
        element[back++] = newdata;
        return true;
    }
    return false;
}
QueueDataType Queue::dequeue() {
    return element[front++]; // actually, isempty() check is necessary
}
bool Queue::isempty(){
    return (front == back);
}
```

Even though the array has space, a “queue full” error might return .

Realization on an Array 3: Circular Array

- This is the proper implementation of a queue on an array.
- We realize the array as a circular array rather than a linear array.
- The first element follows the last element of the array.
- We achieve continuity.
- At different times, there will be elements in different parts of the array.
- As long as the array is not completely full, there will be no "out of space" message.

Circular Array



Circular Array Realization

- An array with max number of elements is used.

Queue[0 ... max - 1]

- A transition from the element with index "max - 1" to the element with index 0 is desired.

```
if (i == max - 1)  
    i = 0;                i = (i + 1) % max  
else  
    i++;
```

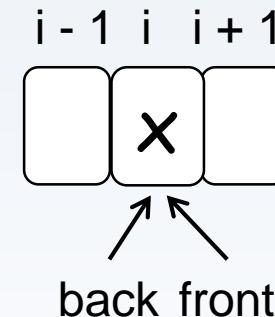
Boundary Conditions

Configuration of front and back pointers:

back: indicates the last element,
increment the "back" pointer, make an assignment.

front: indicates the first element,
do a read, increment the "front" pointer.

If there is a single element in the queue, "front" and "back" pointers will point to this element.



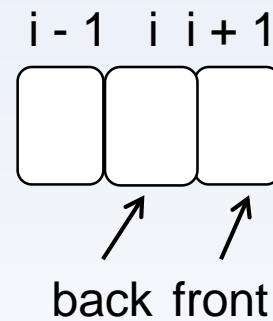
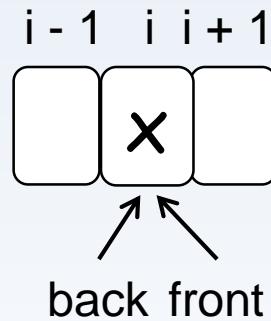
Remember: It is also possible to configure the back and front pointers in different ways.

"back" can point to the empty space instead of the last written element.

Boundary Conditions (continued)

"Queue is empty" condition:

- When this element is removed, the "front" pointer will get incremented once and pass "back".



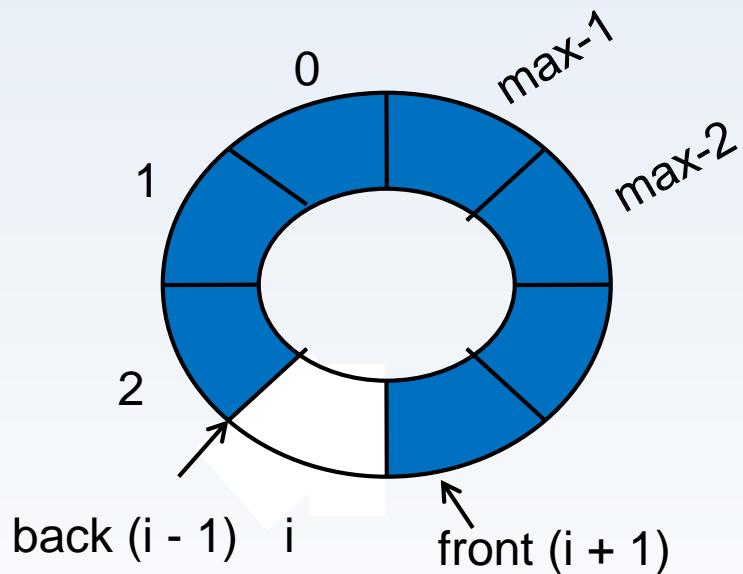
"Queue is empty" condition:

$\text{front} == \text{back} + 1$

Boundary Conditions (continued)

"Queue is full" condition:

- If there is only a single empty slot in the queue, and a new element is added, the "front" will be one more than "back".



"Queue is full" condition: "front" pointer one more than "back"

$\text{front} == \text{back} + 1$

Boundary Conditions (continued)

Empty and Full queue conditions become the same!

Same problem exists even if you use other back and front configurations.

Solutions:

1. Keep array capacity one less. Leave one space empty.
2. Use a logical variable that will indicate that the queue is full.
Or, use a variable that contains the number of elements in the queue.
3. Assign a special value to the pointer to show the state of being empty.

Realization Using Special Index Values

- Initial state conditions: $\text{front} = 0$, $\text{back} = -1$
- When the queue becomes empty, return to initial state conditions.
- If the queue is full, $\text{front} == \text{back} + 1$ and $\text{back} != -1$.

```
void Queue::create() {
    front = 0; back = -1;
}

void Queue::close() {
}

bool Queue::enqueue(QueueDataType newdata) {
    if ( back != -1 && (front == (back + 1) % QUEUESIZE ) )
        return false;
    else {
        back = (back + 1) % QUEUESIZE;
        element[back] = newdata;
        return true;
    }
}
```

```
QueueDataType Queue::dequeue() {
    QueueDataType el = element[front];
    if ( front == back ) {
        front = 0; back = -1;
    }
    else front = (front + 1) % QUEUESIZE;
    return el;
}

bool Queue::isempty() {
    return (front == 0 && back == -1);
}
```

Realization of Queues on Arrays

- Realizations of queues on the array presents problems similar to realization of stacks on the array:
The fixed size of the queue results in an error if we try to add more entries into the queue than expected.
This restriction can be eliminated by using linked lists.
- On the other hand, the implementation on an array may run faster than the implementation with linked lists.

Realization Using Lists

- The restrictions imposed by array implementations can be eliminated by using linked lists.
- In list operations, adding to and removing from the beginning of the list is faster.
 - However, the whole list has to be traversed to find the end of the list.
- To add to the back of the queue, moving each time from the beginning of the list to the end is an expensive operation. Therefore we set the back pointer to point to the last element of the list.

Realization Using Lists – Our Design

- The front of the queue is the first element of the list, the back of the queue is the last element of the list.
- Adding to queue: adding an element to the end of the list
- Removing from the queue: removing an element from the beginning of the list.

```
typedef char QueueDataType;

struct Node{           // nodes of the list
    QueueDataType data;
    Node *next;
};

struct Queue {
    Node *front;
    Node *back;
    void create();
    void close();
    void enqueue(QueueDataType);
    QueueDataType dequeue();
    bool isempty();
};

```

```

void Queue::create(){
    front = NULL; back = NULL;
}
void Queue::close(){
    Node *p;
    while (front) {
        p = front;
        front = front->next;
        delete p;
    }
}
void Queue::enqueue(QueueDataType newdata){
    Node *newnode = new Node;
    newnode->data = newdata;
    newnode->next = NULL;
    if ( isempty() ) {    // first element?
        back = newnode;
        front = back;
    }
    else {
        back->next = newnode;
        back = newnode;
    }
}

```

```

QueueDataType Queue::dequeue() {
    Node *topnode;
    QueueDataType temp;
    topnode = front;
    front = front->next;
    temp = topnode->data;
    delete topnode;
    return temp;
}
bool Queue::isempty() {
    return front == NULL;
}

```

Data Structures

Recursive Programming

Repeating a Code Segment

- There are two basic methods for repeating a code segment:
 - Loops (do-while, while, for, ...)
 - Recursion
- **Recursion**: a function calling itself
 - Simplifies code writing for the solutions to some types of problems.
 - Is an important advanced programmed technique and shortens the code when used appropriately.
 - Has a **downside**: The program may take longer to run.

Recursive Programming

- The basic approach is to define the problem to be solved in terms of simpler subproblems.
- The recursive function can solve the base case and returns the result if there is one.
- If the input problem is not the base case, the problem can be divided into simple parts, and the function calls itself for each part.
- Since recursive program writing is different from the usual style, it takes more practice to learn.

Example: Computing the Factorial

- Definition of the factorial function:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 3 \cdot 2 \cdot 1 & \text{if } n \geq 1 \end{cases}$$

$$n! = n \cdot (n - 1) \cdot \dots \cdot 3 \cdot 2 \cdot 1$$

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

- We could also define the function in terms of itself:

Example: $\text{fact}(5) = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) = 5 \cdot \text{fact}(4)$

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{else} \end{cases}$$

Factorial Function

```
int recursiveFactorial(int n) {  
    int m;  
    if (n == 0) return 1;    // base case  
    else {                  // recursion step  
        m = recursiveFactorial(n - 1);  
        return n * m;  
    }  
}
```

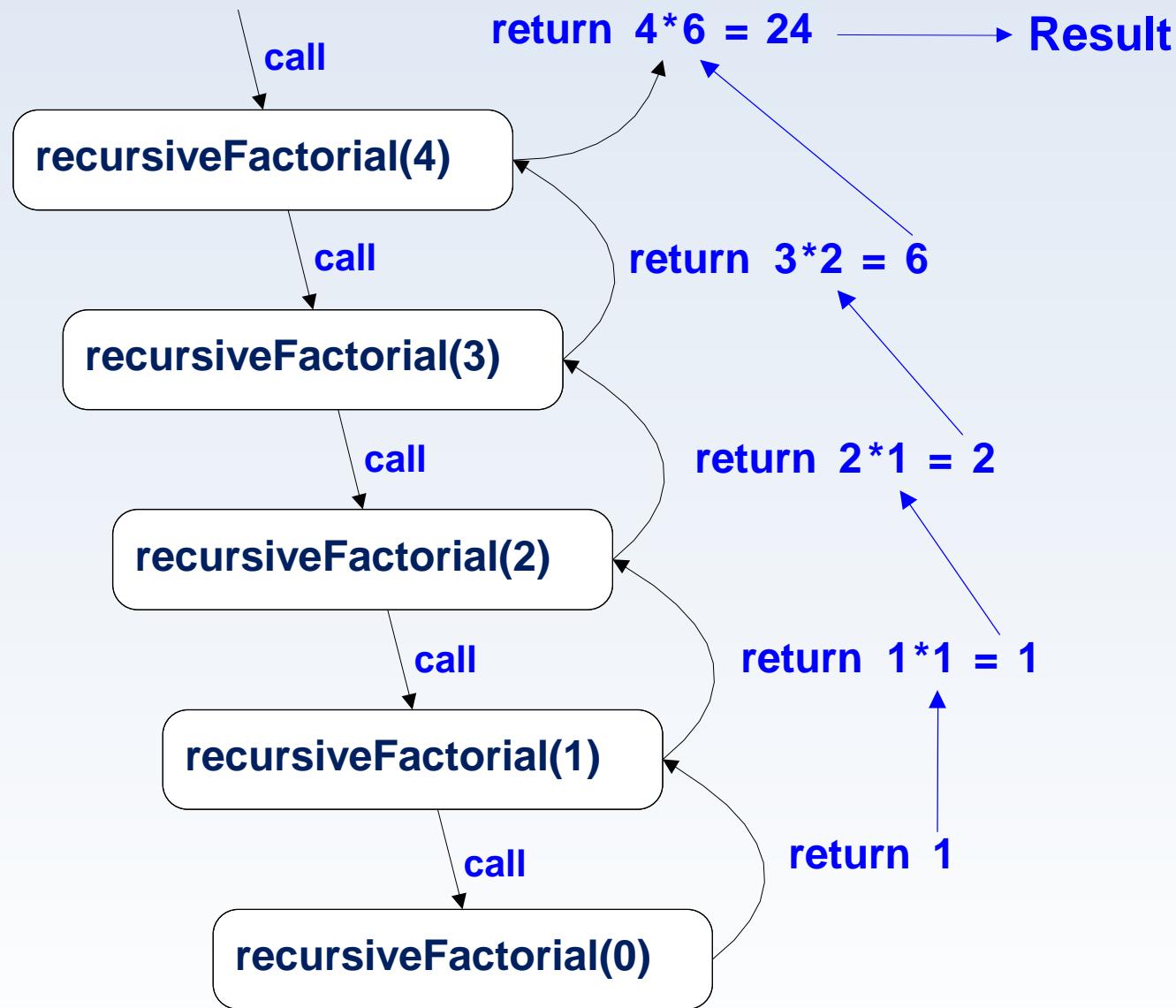
Factorial Function

```
int recursiveFactorial(int n) {  
    int m;  
    if (n == 0) return 1; // base case  
    else { // recursion step  
        return n * recursiveFactorial(n - 1);  
    }  
}
```

Note:

The base case is, at the same time, the termination condition of the function and should not be left out. Otherwise, there will be “infinite” recursion (analogous to the problem of an infinite loop in an iterative solution).

Exercise



Is Recursion Always Useful?

- Let us write the same function iteratively:

```
int iterativeFactorial(int n) {  
    int i, p;  
    p = 1;  
    for (i = 2; i <= n; i++)  
        p *= i;  
    return p;  
}
```

- This solution wastes less space in memory and works faster.

Recursion Types

- **Linear**
 - Each time a function (method) is called, it makes at most one recursive call.
- **Tail**
 - Tail recursion occurs when a linearly recursive function makes its recursive call as its last step.
 - These types of recursive functions can be easily converted to iterative functions.
- **Binary**
 - If the function calls itself twice at a time (i.e., there are two recursive calls for each non-base case), the function uses binary recursion.

Binary Recursion Example

Let us write the function that sums elements of an array.

Algorithm:

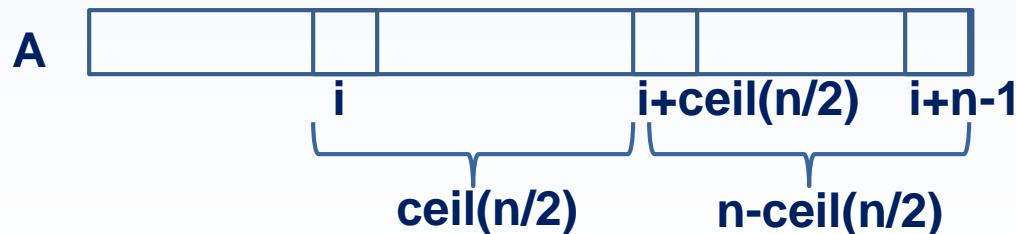
BinarySum(A, i, n)

Input: array A, integers i and n

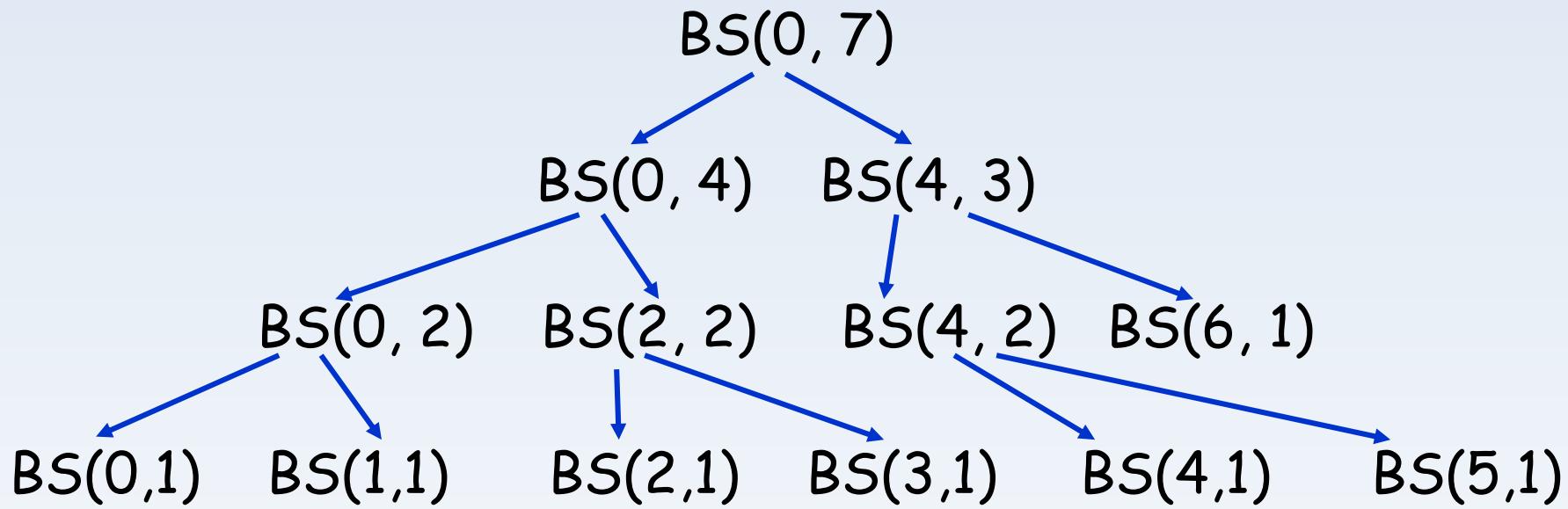
Output: sum of n numbers in array A, starting with index i

```
if (n == 1) then
    return A[i]
```

```
else
    return BinarySum(A, i, ceil(n/2))
        + BinarySum(A, i+ceil(n/2), n-ceil(n/2))
```



BinarySum() Call Sequence



- Binary recursion is generally used in algorithms designed with the “**divide and conquer**” method.
- In these types of algorithms, recursion may prove more effective than an iterative solution.

Permutation

- Example: permutations of the set {1, 2, 3} :

1 2 3	1 3 2
2 1 3	2 3 1
3 1 2	3 2 1

- To generate these permutations, we could use:

```
for (i1 = 1; i1 <= 3; i1++) {  
    for (i2 = 1; i2 <= 3; i2++) {  
        if (i2 != i1) {  
            for (i3 = 1; i3 <= 3; i3++) {  
                if ( (i3 != i1) && (i3 != i2) ) {  
                    cout << i1 << " " << i2 << " " << i3 << endl;  
                }  
            }  
        }  
    }  
}
```

- However, this code can only generate ternary (3-ary) permutations. How do we generate n-ary permutations?

Recursive Solution for Permutation

```
void perm(int *list, int k, int m) {  
    int i;  
    if (k == m) { // just print out list  
        for (i = 0; i <= m; i++) {  
            cout << list[i] << " ";  
        }  
        cout << endl;  
    }  
    else {  
        for (i = k; i <= m; i++) {  
            swap(&list[k], &list[i]);  
            perm(list, k + 1, m);  
            swap(&list[k], &list[i]);  
        }  
    }  
}
```

list[0..x] -> an array storing the data that can be printed to the screen
k: starting index
m: ending index
First call: perm(list, 0, x)
Ex: perm(list,0,3) for an array of size 4 elements

```
void swap(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

Recursive Solution for Permutation

- At each step, a position is held constant, and the others are permuted.
- For example, when writing the permutation of (1 2 3 4), these numbers are placed into the first position in order, and the remaining three positions are written as a permutation.
 - 1 ... permutations that start with 1
 - 2 ... permutations that start with 2
 - 3 ...
 - 4 ...
- Thus, the problem is divided into subproblems.

Eight Queens Problem

- A classical problem
- **Goal:** Place 8 queens on a 8×8 chessboard such that they cannot attack each other
- Two queens cannot be located on the
 - Same row
 - Same column
 - Same diagonal
- A solution for a 5×5 board is shown on the right.
 - At the top of every column, the number of the row where the queen is located is shown.

5	3	1	4	2
		q		
				q
		q		
				q
q				

Solution to the Eight Queens Problem

- The problem has more than one solution.
- Since only one queen may be located on a row, the potential solutions may be found by generating permutations of the row numbers shown on the example board.
 - All permutations may not be solutions.
 - However, all solutions are among these permutations.

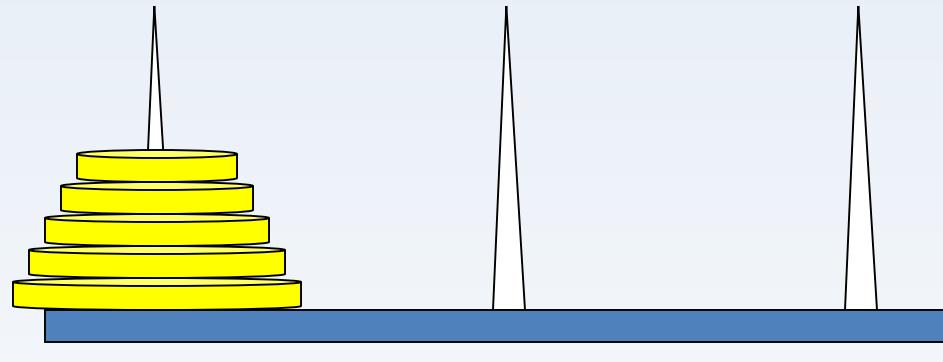
4	1	3	5	2
	q			
				q
			q	
q				
				q

Solution to the Eight Queens Problem

- A permutation generator with the addition of “diagonal attack” checks could be used to solve the *n queens problem*.

Towers of Hanoi

- The Towers of Hanoi problem can easily be solved using recursion.



stacks

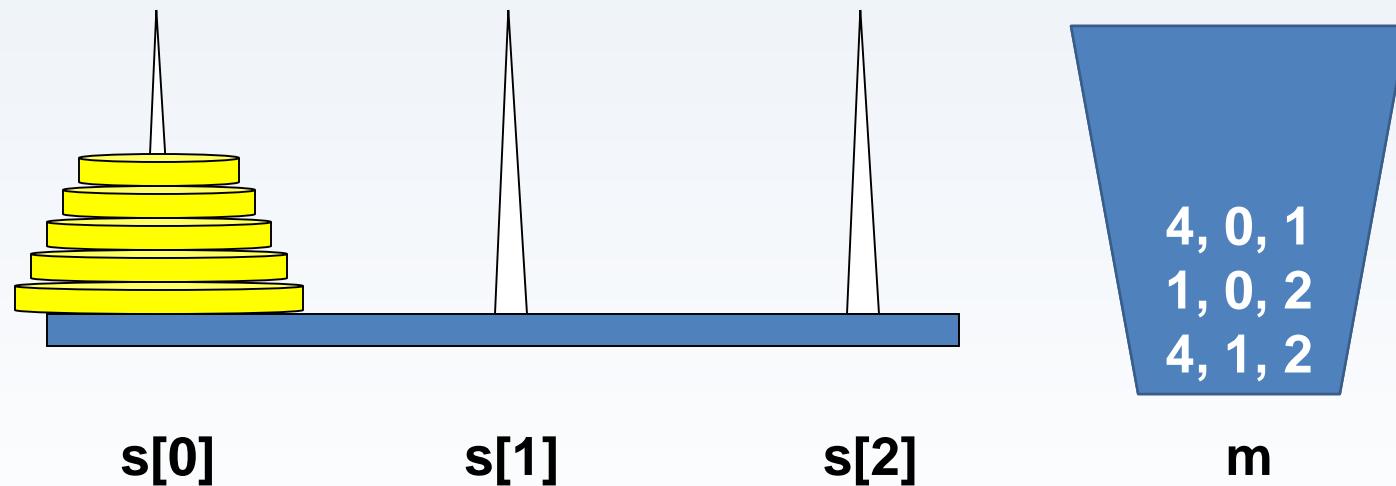
s[0]

s[1]

s[2]

Hanoi Iterative Solution (reminder)

- As long as the move stack is full, we pop a move (n, source, destination) from this stack, and push these moves onto the stack:
 - (n - 1, temp, destination)
 - (1, source, destination)
 - (n - 1, source, temp) (what has to be done first is at the top of the stack)



Hanoi Recursive Solution

```
void Hanoi_recursive(int n, int source, int destination, int temp) {  
    if (n >= 1) {  
        Hanoi_recursive(n - 1, source, temp, destination);  
        s[destination].push( s[source].pop() );  
        Hanoi_recursive(n - 1, temp, destination, source);  
    }  
}  
  
Hanoi_recursive(5, 0, 2, 1);
```

Finding a Path in a Labyrinth

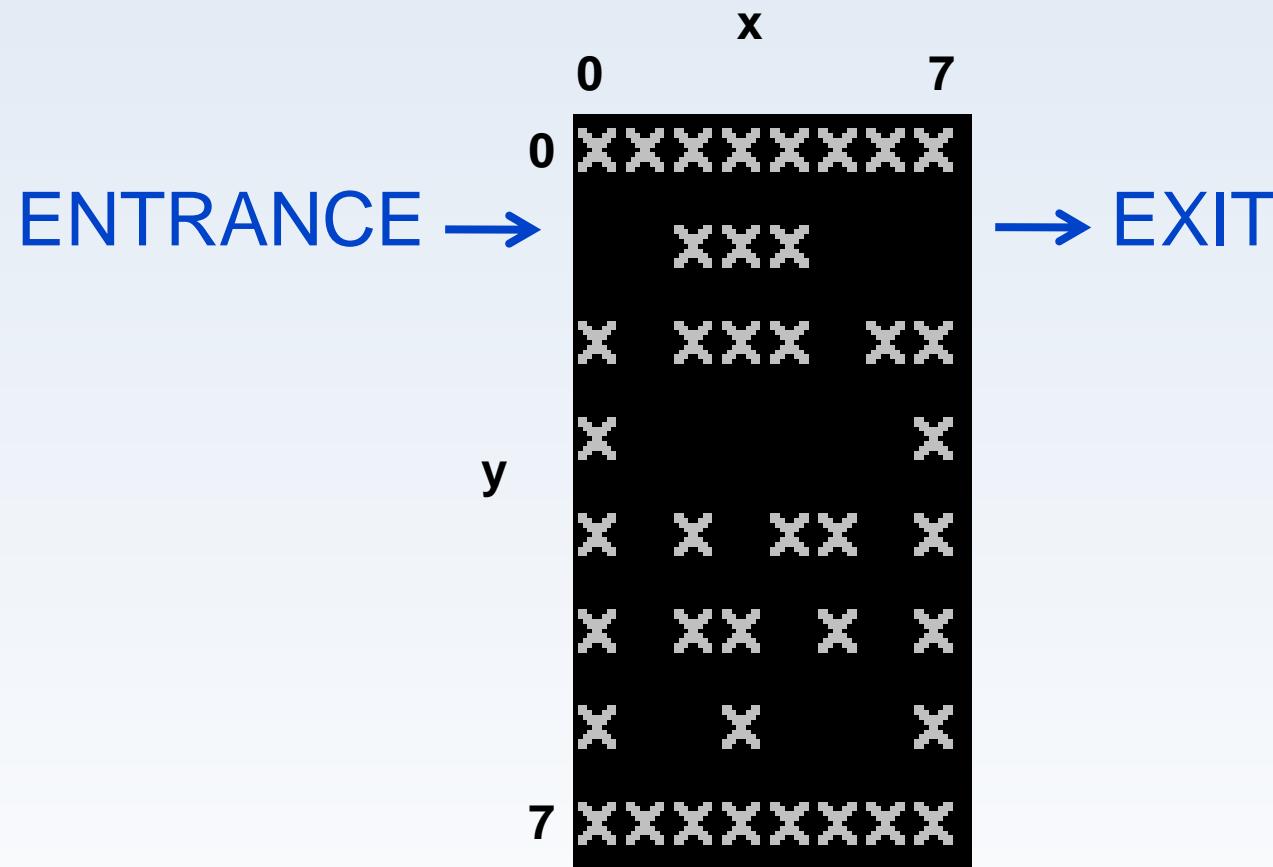
- The problem of finding a path in a labyrinth, which we had previously solved using a stack, could also be solved by writing a recursive function.

```
int main(){
    char lab[8][8] = {{'x','x','x','x','x','x','x','x'},
                      {' ',' ',' ','x','x','x',' ',' ',' '},
                      {'x',' ',' ','x','x','x',' ','x','x'},
                      {'x',' ',' ',' ',' ',' ',' ',' ','x'},
                      {'x',' ',' ','x',' ','x','x',' ','x'},
                      {'x',' ',' ','x',' ','x','x',' ','x'},
                      {'x',' ',' ','x','x','x',' ','x'},
                      {'x',' ',' ',' ','x',' ',' ',' ','x'},
                      {'x','x','x','x','x','x','x','x'}};

    if ( find_path(lab, entrance.y, entrance.x, LEFT) )
        cout << "PATH found" << endl;

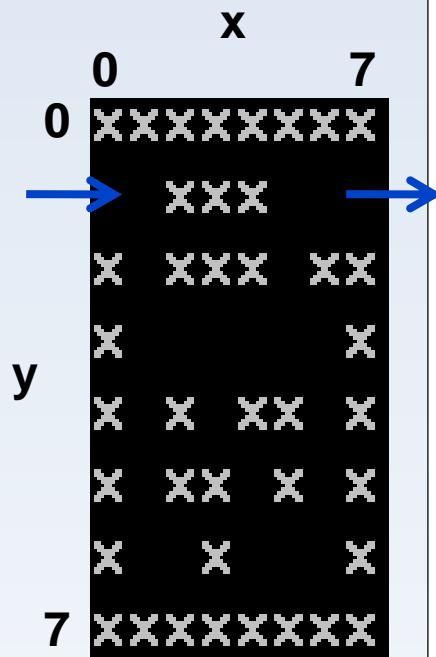
    printlab(lab);
    return EXIT_SUCCESS;
}
```

Finding a Path in a Labyrinth



Recursive Path Finding Function

```
bool find_path(char lab[8][8], int y, int x, int camefrom) {  
    lab[y][x] = 'o';  
    if ( x == lab_exit.x && y == lab_exit.y )  
        return true;  
    printlab(lab);  
    if ( x > 0 && lab[y][x - 1] != 'x' && camefrom != LEFT )  
        if ( find_path(lab, y, x - 1, RIGHT) ) // go left  
            return true;  
    if ( y < 7 && lab[y + 1][x] != 'x' && camefrom != DOWN )  
        if ( find_path(lab, y + 1, x, UP) ) // go down  
            return true;  
    if ( y > 0 && lab[y - 1][x] != 'x' && camefrom != UP )  
        if ( find_path(lab, y - 1, x, DOWN) ) // go up  
            return true;  
    if ( x < 7 && lab[y][x + 1] != 'x' && camefrom != RIGHT)  
        if ( find_path(lab, y, x + 1, LEFT) ) // go right  
            return true;  
    lab[y][x] = ' '; // delete incorrect paths  
    printlab(lab); // display the return from the incorrect paths as well  
    return false;  
}
```



Finding a Path in a Labyrinth

xxxxxxxx
o xxx
x xxx xx
x x
x x xx x
x xx x x
x x x
xxxxxxxx

xxxxxxxx
ooxxx
xoxxx xx
x x
x x xx x
x xx x x
x x x
xxxxxxxx

xxxxxxxx
ooxxx
xoxxx xx
xo x x
xo x xx x
x xx x x
x x x
xxxxxxxx

xxxxxxxx
ooxxx
xoxxx xx
xo x
xo x xx x
xo xx x x
xo x x
xxxxxxxx

xxxxxxxx
ooxxx
xoxxx xx
xo
xo x xx x
xo xx x x
xo x x
xxxxxxxx

xxxxxxxx
ooxxx
x xxx xx
x x
x x xx x
x xx x x
x x x
xxxxxxxx

xxxxxxxx
ooxxx
xoxxx xx
xo x x
xo x xx x
x xx x x
x x x
xxxxxxxx

xxxxxxxx
ooxxx
xoxxx xx
xo x x
xo x xx x
xo xx x x
xo x x
xxxxxxxx

xxxxxxxx
ooxxx
xoxxx xx
xo x
xo x xx x
xo xx x x
xo x x
xxxxxxxx

xxxxxxxx
ooxxx
xoxxx xx
xo
xo x xx x
xo xx x x
xo x x
xxxxxxxx

1

3

5

7

9

2

4

6

8

10

Finding a Path in a Labyrinth

xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
ooxxx	ooxxx	ooxxx	ooxxx	ooxxx	ooxxx
xoxxx xx					
xo x x	xoo x	xooo x	xooo x	xooo x	xooo x
xox xx x	x x xx x	x xox x	x x xx x	x x xx x	x x xx x
x xx x x					
x x x	x x x	x x x	x x x	x x x	x x x
xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
ooxxx	ooxxx	ooxxx	ooxxx	ooxxx	ooxxx
xoxxx xx					
xo x x	xooo x				
x x xx x					
x xx x x					
x x x	x x x	x x x	x x x	x x x	x x x
xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx

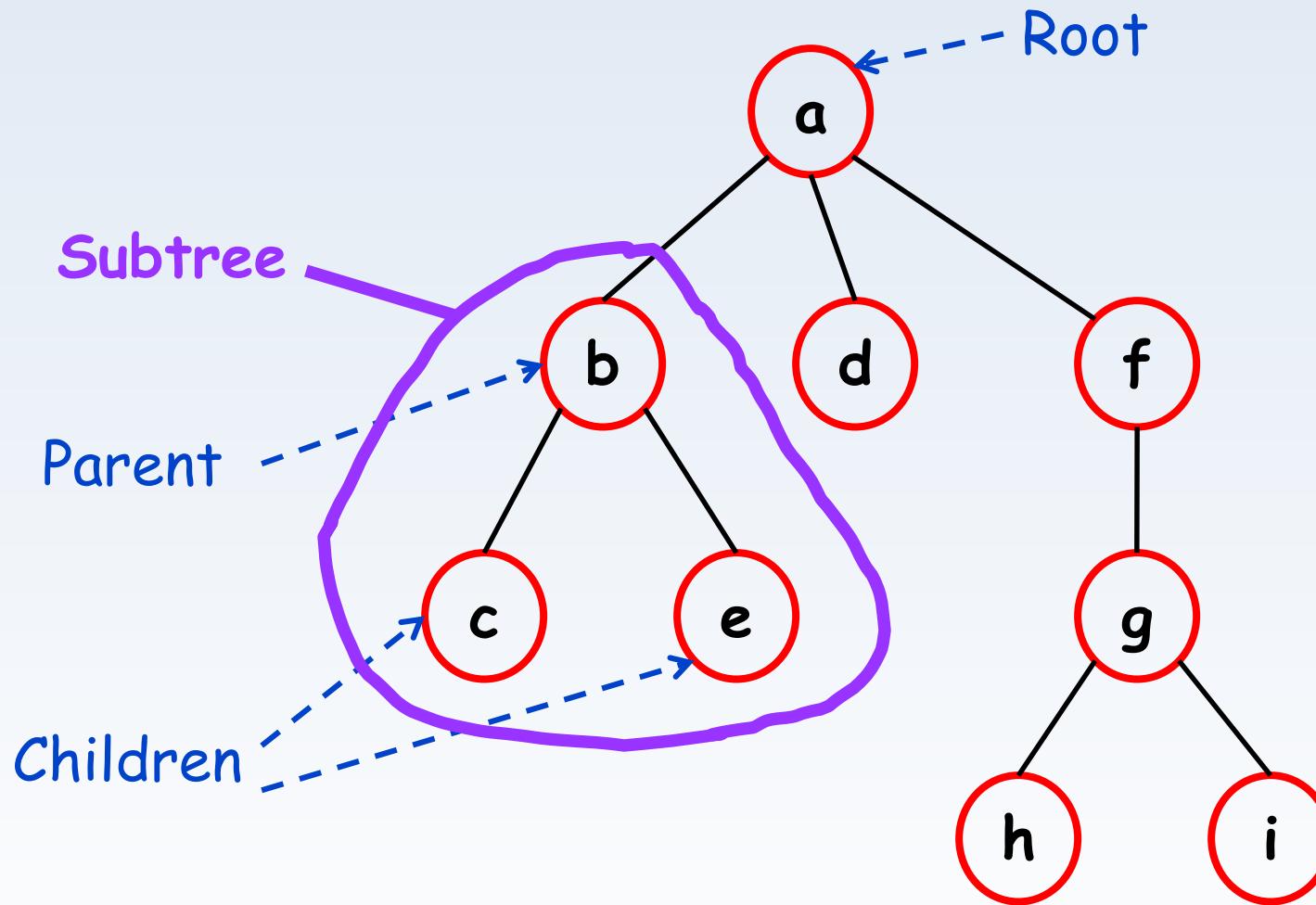
Data Structures

Trees

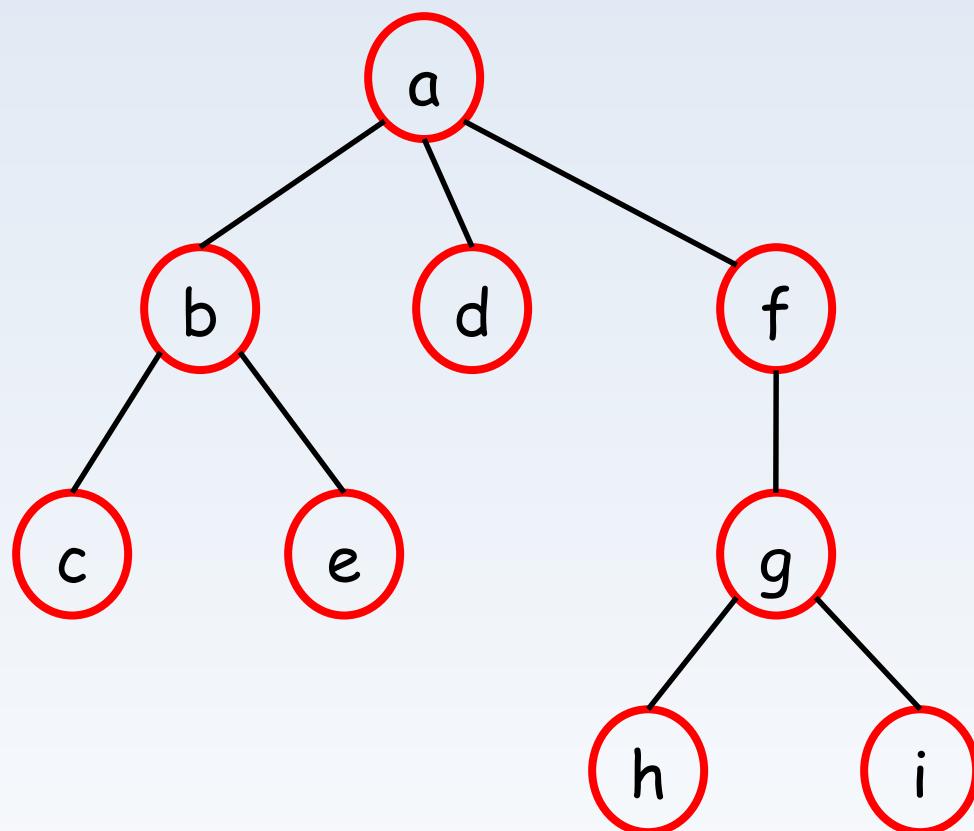
Tree Definition

- The tree is a finite set of nodes.
 - There is a special node called the **root**.
 - The remaining nodes make up n separate, disjoint sets.
 - Each set has a separate tree structure.
 - These sets are called **subtrees**.
- The nodes one level below each node are the **child** nodes of that node.
- The node located one level above a node on the way to the root is the **parent** node.

Example



Definitions



Degree: the number of subtrees the root has in a subtree or tree.

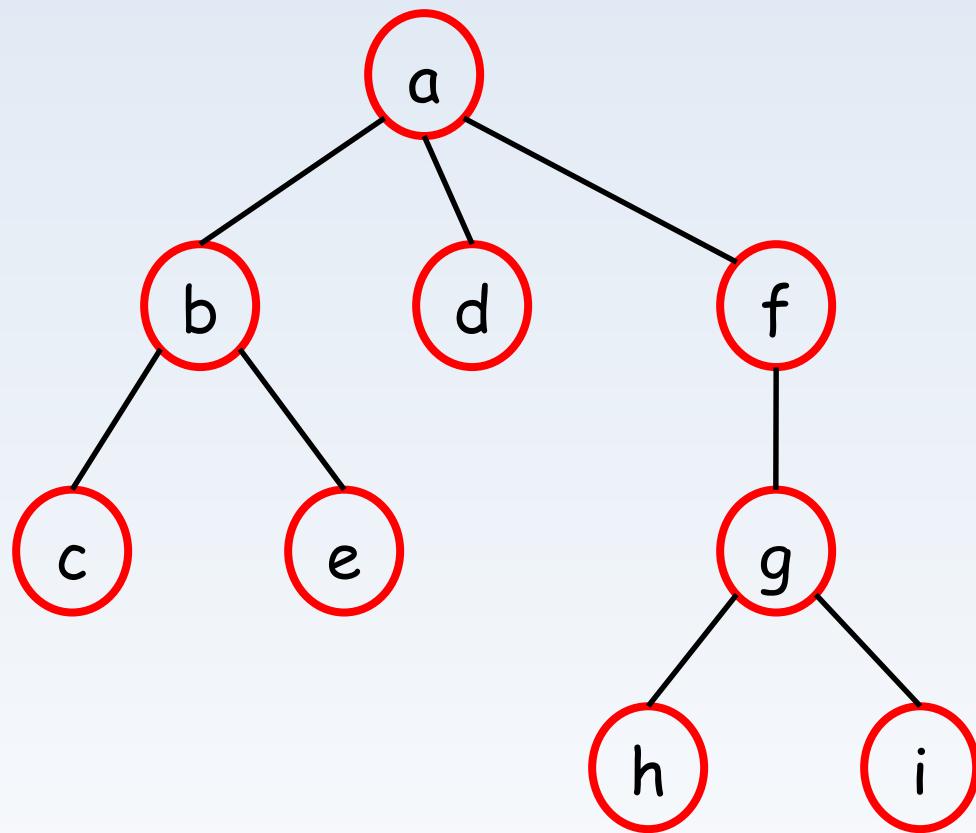
Example:

- $a \rightarrow 3$
- $f \rightarrow 1$

Leaf: Node with degree 0.

Example: c, d, e, h, i

Definitions



Level: Root is assumed to be at level 1 and the level increases down the tree.

Example:

- b: 2
- h: 4

Depth: The depth of the tree is the maximum node level.

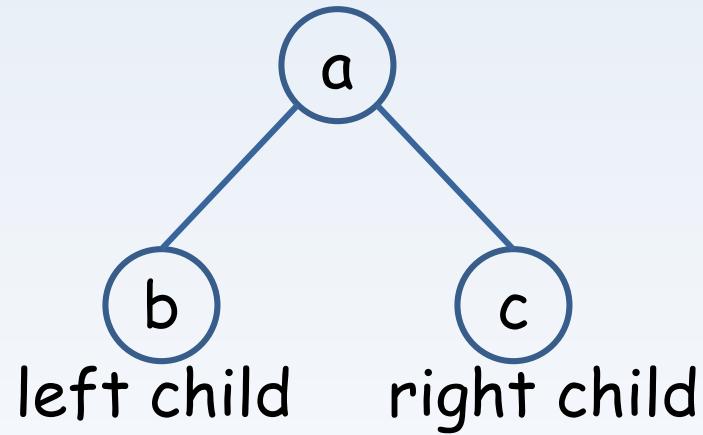
Depth of example tree: 4

Why a Tree?

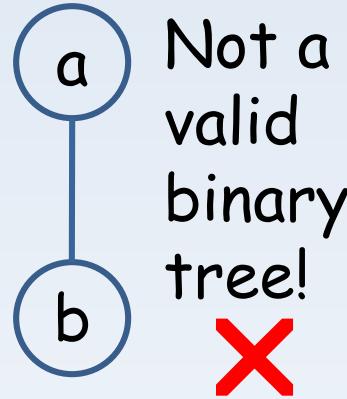
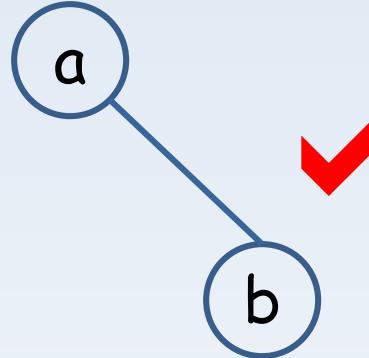
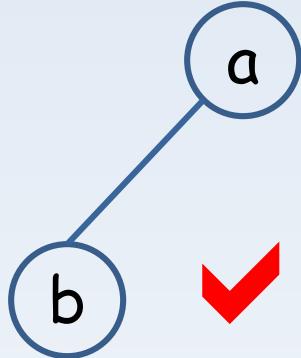
- The list structure is not suited to storing and accessing large amounts of data.
 - Especially if fast access is desired.
- Many operations can be realized with $O(\log n)$ complexity on the tree structure.
 - For now, it suffices for us to know that tree operations, on average, can be performed much faster than list operations.
- Due to this speedup, tree structures are used in databases and many areas where indexing is necessary.
- In this course, we will study in detail the binary search tree, which is the simplest tree structure.
- You will learn about different tree structures in Advanced Data Structures.

Binary Tree

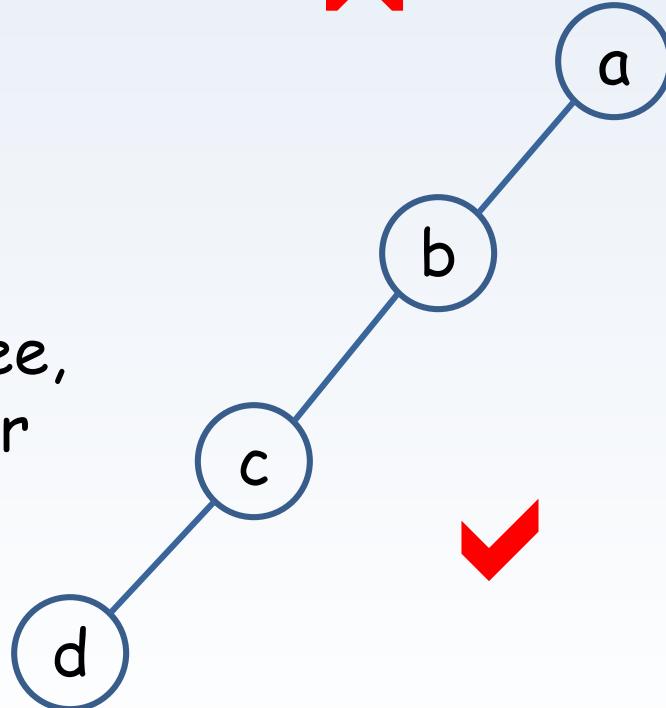
- Every node has at most two subtrees.
 - The degree of a node cannot be larger than 2.
- Subtrees are called left and right subtrees.



Binary Tree Examples



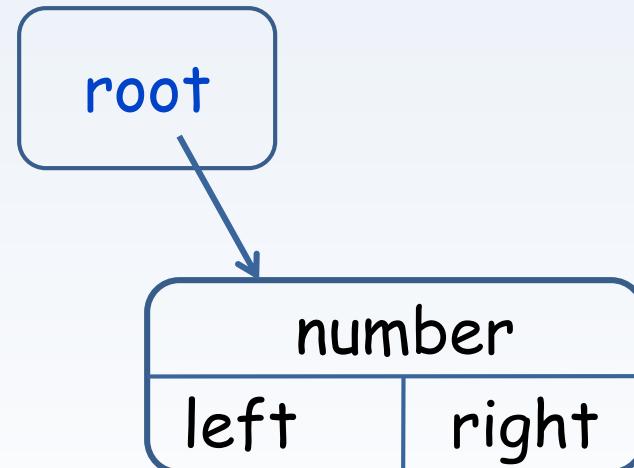
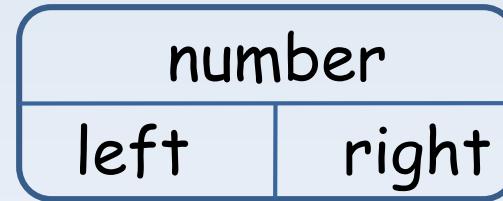
- Left and right subtrees are interpreted differently.
- Even if a node has a single subtree, this subtree's being on the left or the right changes the tree.



Data Structure

```
struct node {  
    int number;  
    node *left;  
    node *right;  
};
```

```
struct tree {  
    node *root;  
};
```



Using the Tree Type

- We must first create an empty tree.

```
tree directory;
```

```
directory.root = NULL;
```

- Let us create a new node. Let the number in the node be "2".

```
node *p;
```

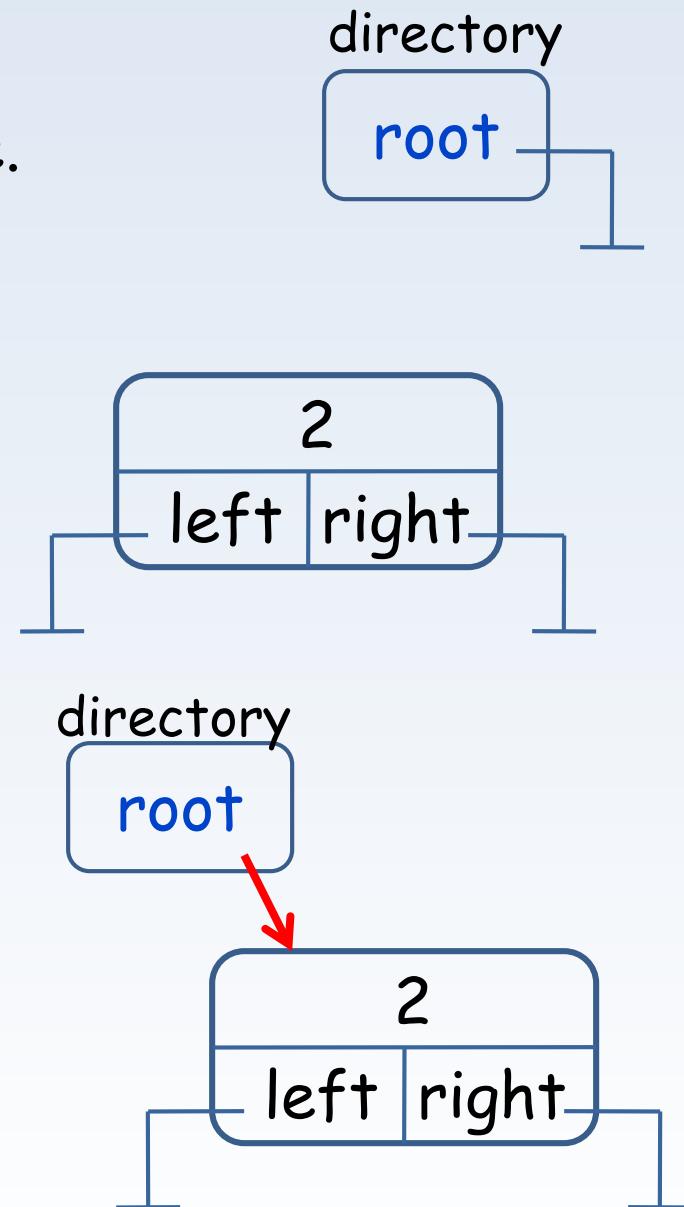
```
p = new node;
```

```
p->number = 2;
```

```
p->left = p->right = NULL;
```

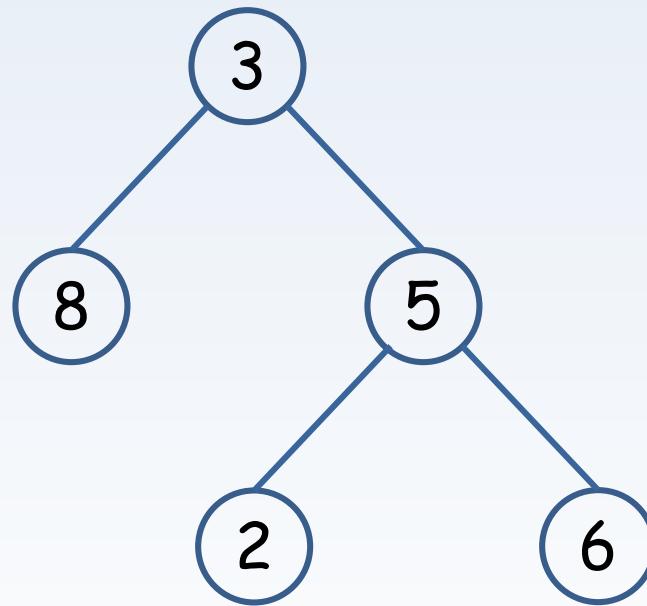
- Let us add this node as the root to our "directory".

```
directory.root = p;
```



Tree Representation

- From this point forward, we will not show left and right pointers in the diagrams. These always exist at each node.
- A simpler diagram:



Tree Traversal

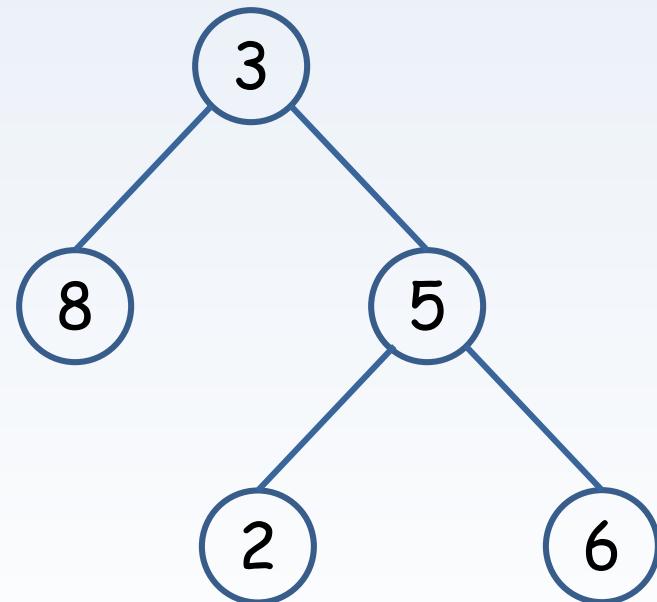
- Traversing a binary tree by visiting all nodes can be done in three ways:
 - Preorder
 - Inorder
 - Postorder
- All three types of traversals are, by definition, recursive operations.

Preorder

- Traversal order
 - First, node itself
 - Then, left subtree
 - Finally, right subtree
- This operation repeats for each node.

Example:

3 8 5 2 6

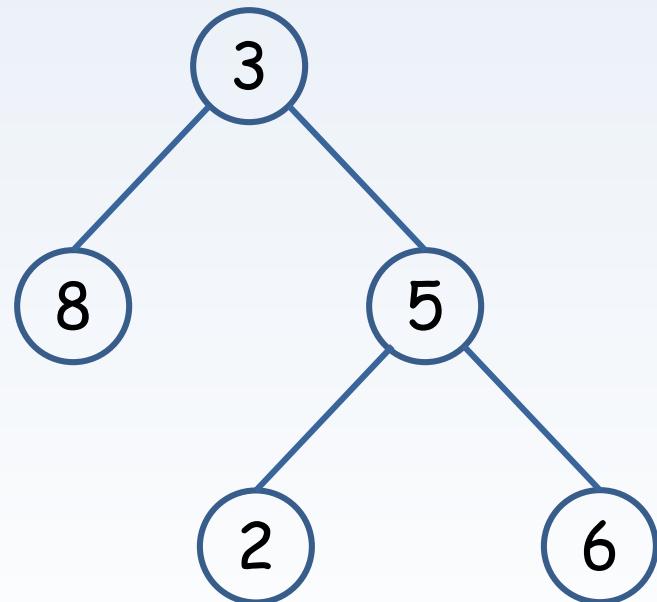


Inorder

- Traversal order
 - First, left subtree
 - Then, node itself
 - Finally, right subtree
- This operation repeats for each node.

Example:

8 3 2 5 6

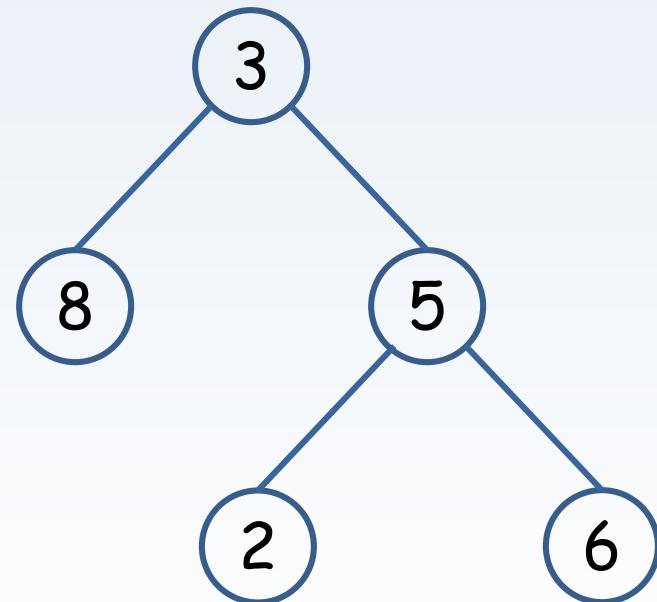


Postorder

- Traversal order
 - First, left subtree
 - Then, right subtree
 - Finally, node itself
- This operation repeats for each node.

Example:

8 2 6 5 3



Coding the Traversal Methods

- All three traversal methods can easily be programmed recursively.
- Let us assume that the previously defined structure called "directory" has been created.
- Let us write the traversal functions that will work on this structure and print the data in the nodes to the screen.

Preorder Function

```
void Preorder(node *nptr) {  
    if (nptr) {  
        cout << nptr->number << endl;  
        Preorder(nptr->left);  
        Preorder(nptr->right);  
    }  
}
```

Inorder Function

```
void Inorder(node *nptr) {  
    if (nptr) {  
        Inorder(nptr->left);  
        cout << nptr->number << endl;  
        Inorder(nptr->right);  
    }  
}
```

Postorder Function

```
void Postorder(node *nptr) {  
    if (nptr) {  
        Postorder(nptr->left);  
        Postorder(nptr->right);  
        cout << nptr->number << endl;  
    }  
}
```

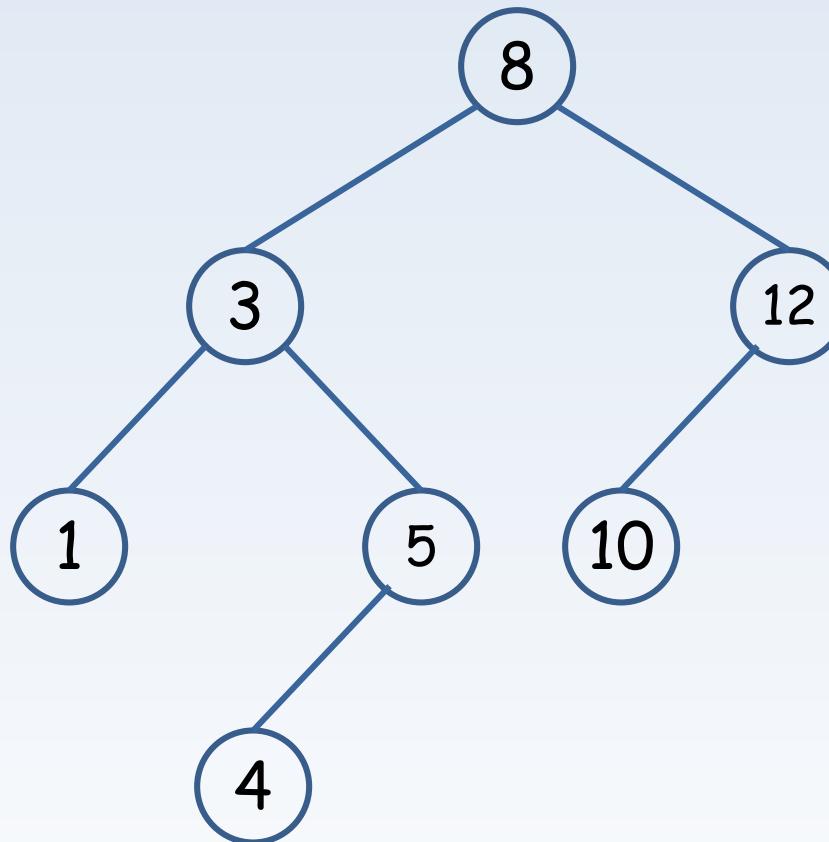
Iterative Inorder Function

```
void Inorder(node *root) {  
    node *current;  
    char flag = 1;  
    stack s;  
    s.create();  
    current = root;  
    while (flag) {  
        while (current != NULL) {  
            s.push(current);  
            current = current->left;  
        }  
        if ( !s.isempty() ) {  
            current = s.pop();  
            cout << current->number  
                << endl;  
            current = current->right;  
        }  
        else  
            flag = 0;  
    }  
}
```

Iterative Inorder Example

- Let us solve using the stack:

1 3 4 5 8 10 12

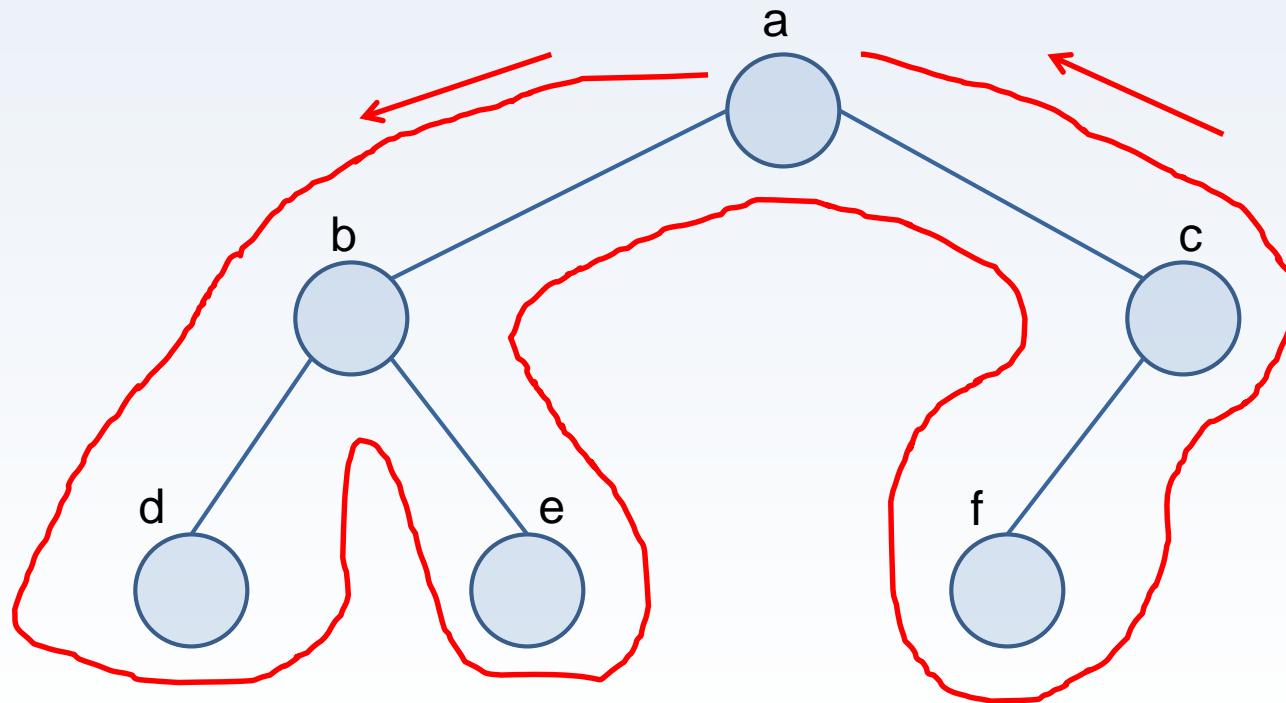


Data Structures

Trees

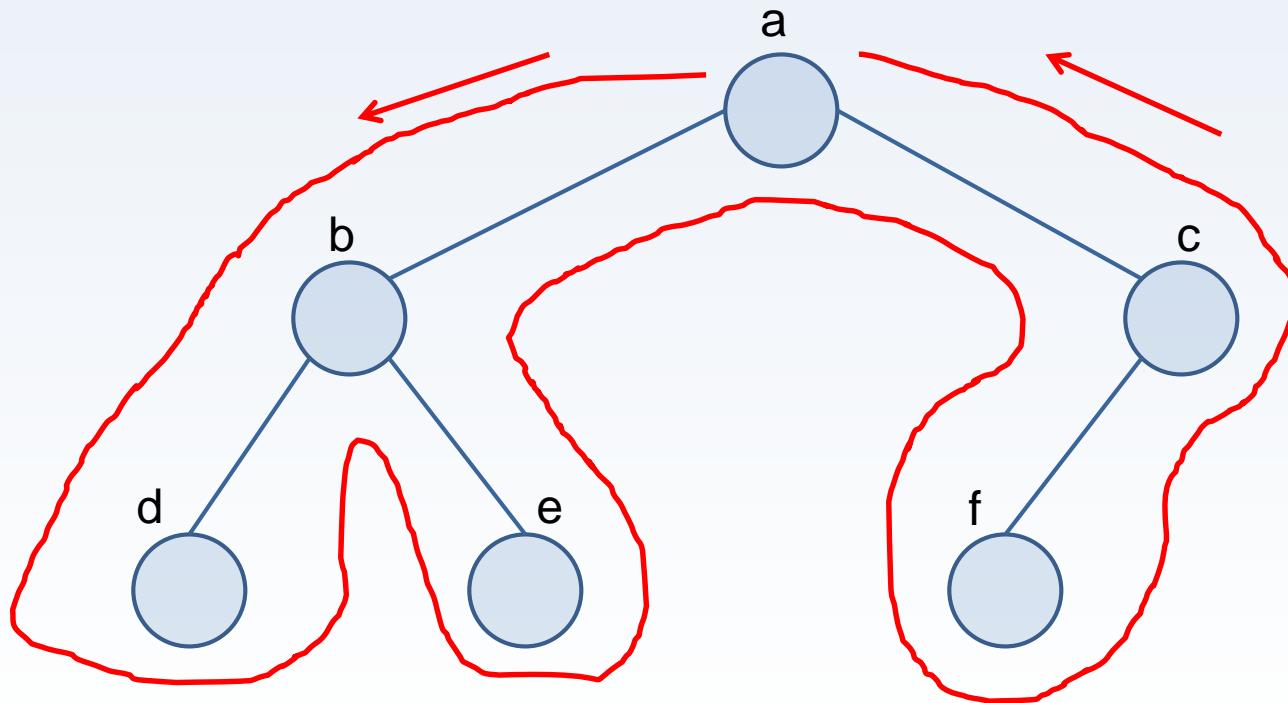
Traversal Orders

- Imagine having a string shrink-wrapped around the tree.
- If we start at the root node and follow the string around the tree, we are **traversing** the tree.
- In what follows, we will always go around the tree counterclockwise.



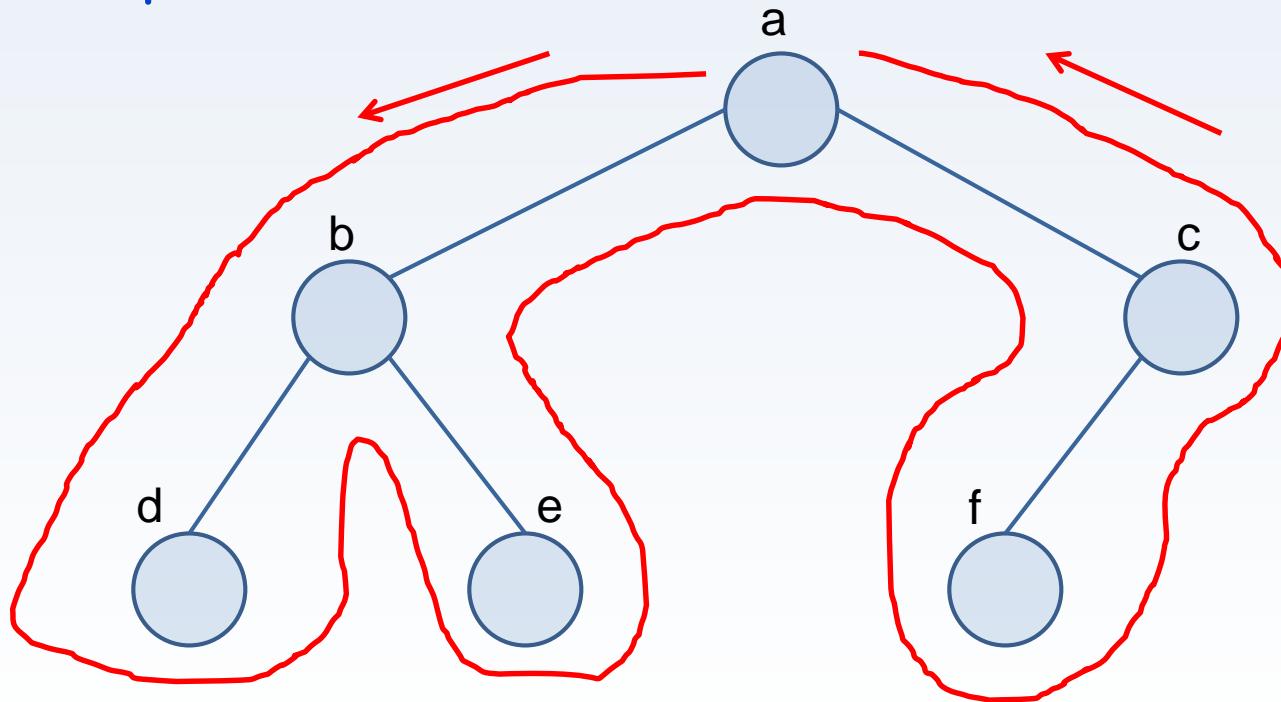
Traversal Orders

- When we traverse the tree, each node is visited three times: on the left, underneath, and on the right.
 - To picture this for nodes with one or no children, just attach imaginary children.

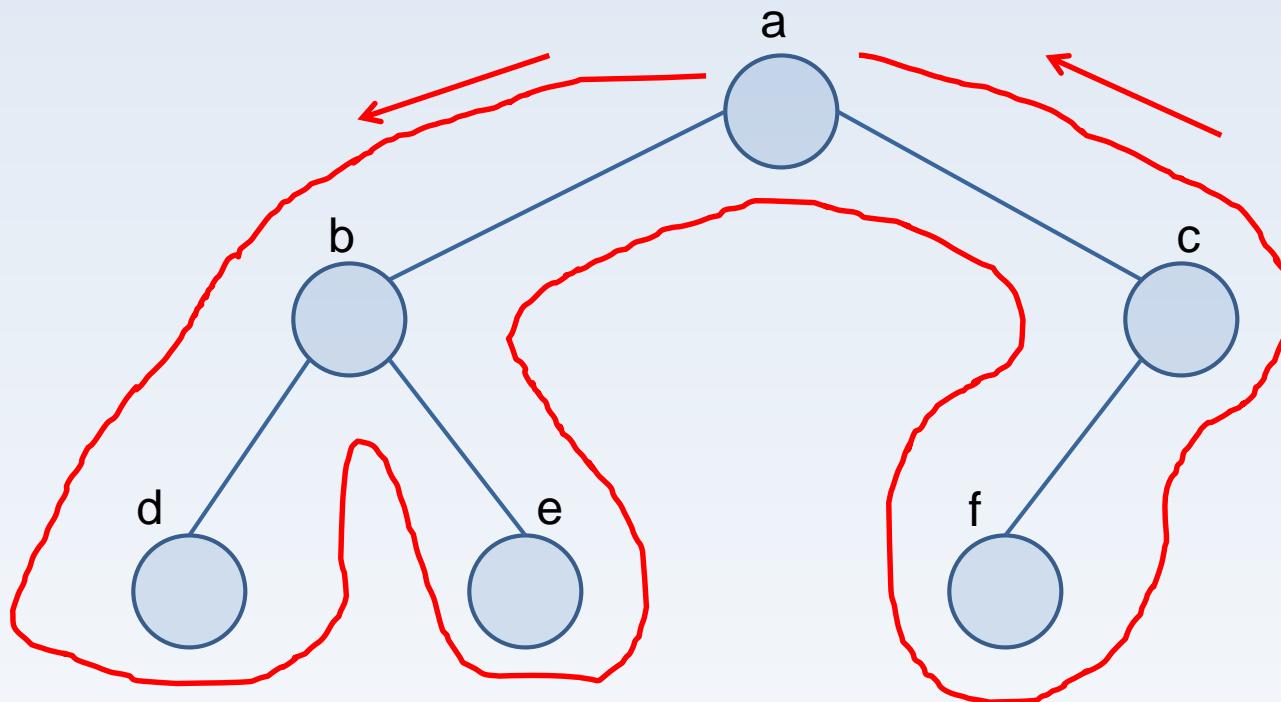


Traversal Orders

- The order of the nodes as they are passed **on the left** is called **preorder**.
- The order of the nodes as they are passed **underneath** is called **inorder**.
- The order of the nodes as they are passed **on the right** is called **postorder**.



Traversal Orders: Example



- Preorder: a b d e c f
- Inorder: d b e a f c
- Postorder: d e b f c a

Binary Search Tree

- Trees are generally used in index structures.
- To use them for this purpose, data should be placed in the tree in a certain order.
- The simplest ordering is the binary search tree.

Rule:

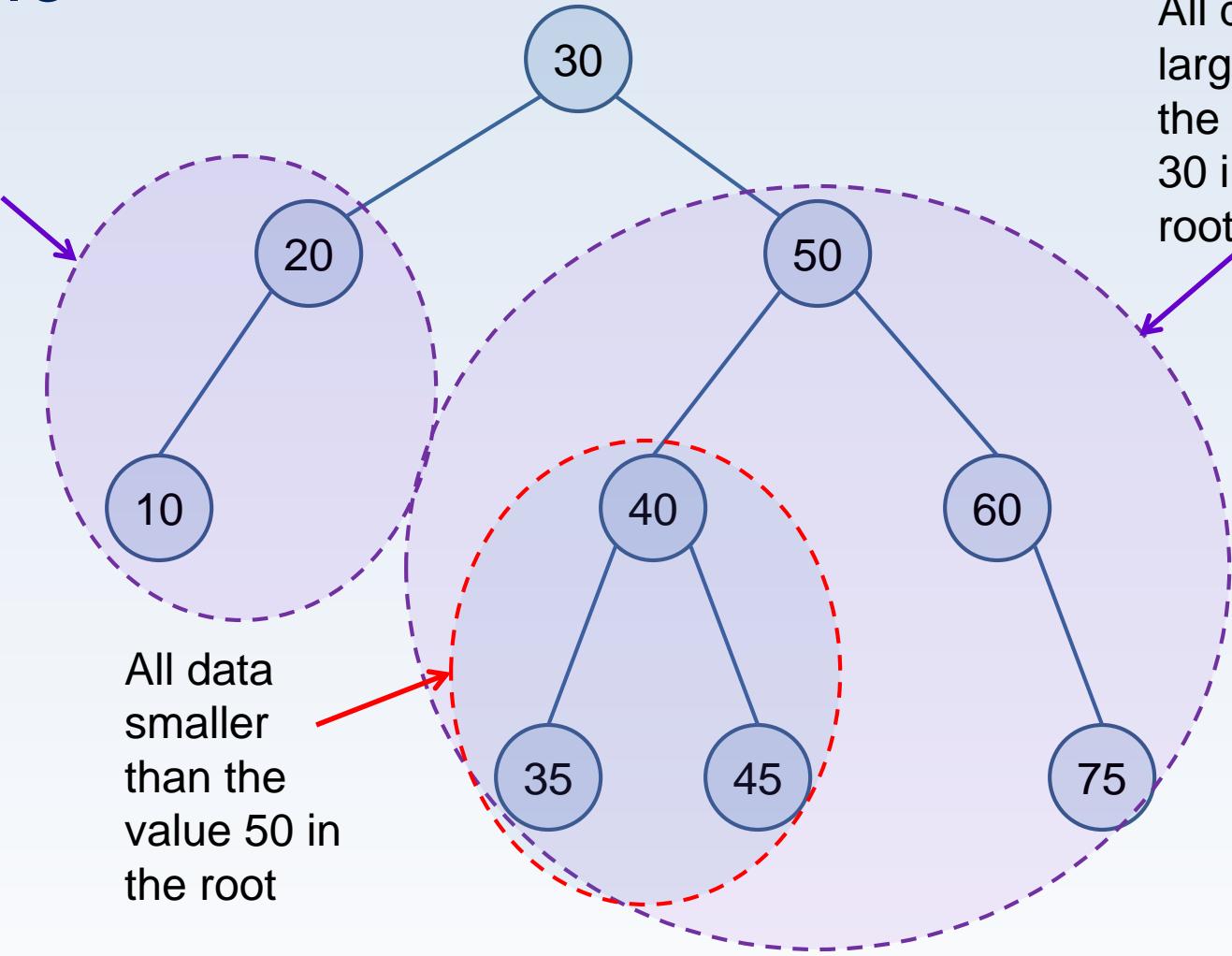
- A root node
 - values in the **left subtree** should be **smaller** than the value in the root
 - values in the **right subtree** should be **larger** than the value in the root.
- The right and left subtrees of a root node have the **search tree property**.

Example

10 and 20
smaller
than the
value 30 in
the root

All data
larger than
the value
30 in the
root

All data
smaller
than the
value 50 in
the root



Note!

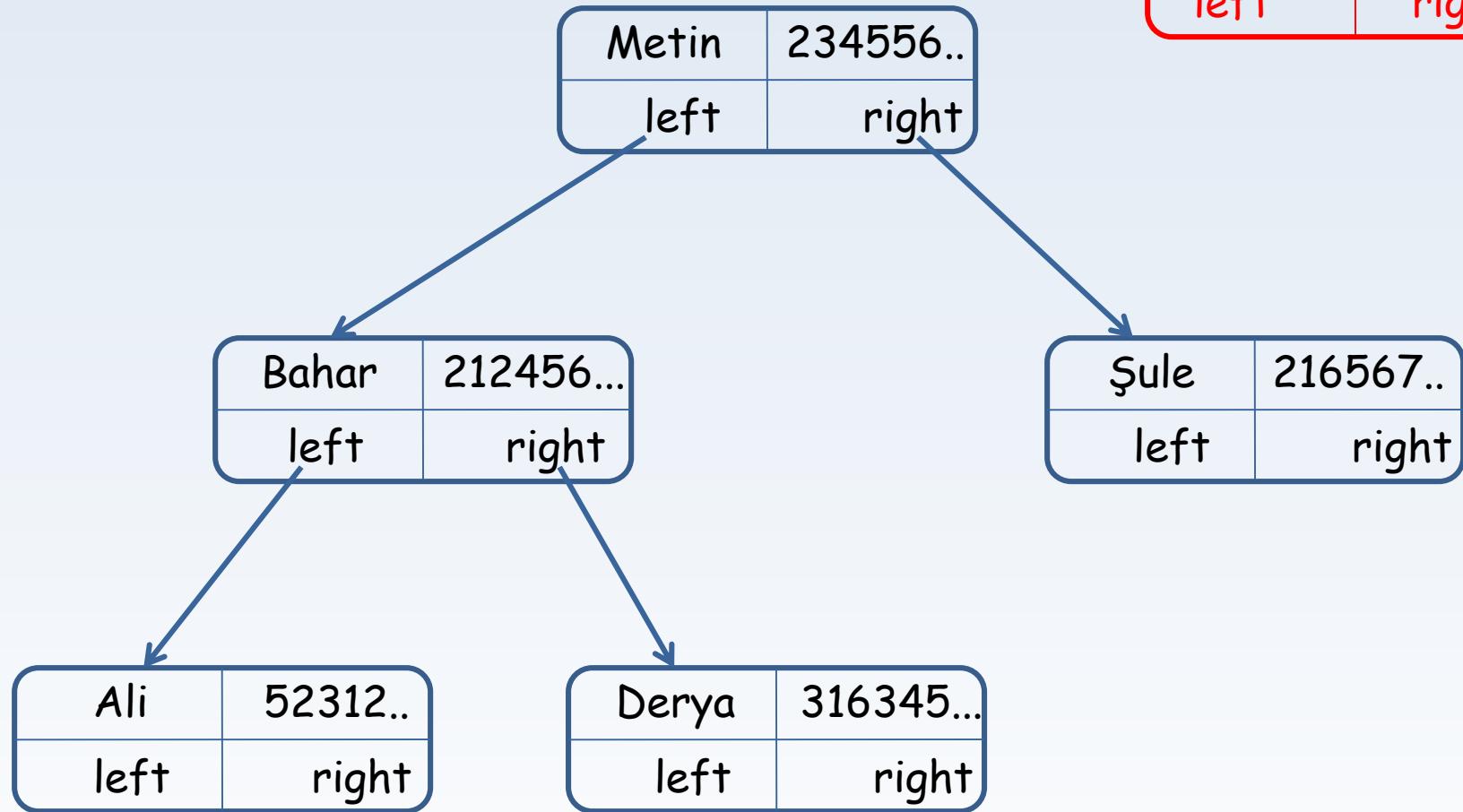
In the binary search tree, each data is located in only 1 node.

isBST(node *nptr){



```
    node *t;
    if(nptr==null) return true;
    if(isBST(nptr->left)){ t=nptr->left;
        if(t!=null) {
            while(t->right) t=t->right;
            if(t->data>=nptr->data) return false;
        }
    }
    else return false;
    if(isBST(nptr->right)){t=nptr->right;
        if(t!=null) {
            while(t->left) t=t->left;
            if(t->data<=nptr->data) return false;
        }
    }
    else return false;
    return true;
}
```

Phone Book



Node Structure

```
// node.h
#define NAME_LENGTH 30
#define PHONENUM_LENGTH 15

struct Phone_node{
    char name[NAME_LENGTH];
    char phonenum[PHONENUM_LENGTH];
    Phone_node *left;
    Phone_node *right;
};
```

Node Structure

```
// tree.h
struct Tree {
    Phone_node *root;
    int nodecount;
    char *filename;
    FILE *phonebook;
    void create();
    void close();
    void emptytree(Phone_node *);
    void add(Phone_node *);
    void remove(char *);
    void remove(Phone_node **);
    void traverse_inorder(Phone_node *);
    int search(char *);
    //void update(int recordnum);
    void read_fromfile();
    void write_inorder(Phone_node *);
    void write_tofile();
};
```

Main Program

- Global definition (in phoneprog.cpp):
`typedef Tree Datastructure;`
- `main()`, `print_menu()`, and `perform_operation()` functions do not need to change.

Constructing the Tree

```
// tree.cpp
void Tree::create() {
    root = NULL; // create empty tree
    nodecount = 0; // initialize nodecount to 0
    read_fromfile();
}
```

- We must make some changes to the function that reads the data from the file.
 - Every record read must be added to the tree in order.

Reading from File and Placing into Tree

```
// tree.cpp
void Tree::read_fromfile() {
    struct File_Record {
        char name[NAME_LENGTH];
        char phonenum[PHONENUM_LENGTH];
    };
    File_Record record;
    Phone_node *newnode;
    filename = "phonebook.txt";
    if (!(phonebook =
        fopen( filename, "r+" )))
        if (!(phonebook =
            fopen( filename, "w+" )))
            cerr << "Could not open file."
            << endl;
        cerr << "Will work in"
            << " memory only."
            << endl;
    return;
}

fseek(phonebook, 0, SEEK_SET);

while (!feof(phonebook)) {
    newnode = new Phone_node;
    fread(&record,
        sizeof (File_Record),
        1, phonebook);
    if ( feof(phonebook) )
        break;
    strcpy(newnode->name, record.name);
    strcpy(newnode->phonenum,
        record.phonenum);
    newnode->left = newnode->right =NULL;
    add(newnode);
    delete newnode;
}
fclose(phonebook);
}
```

Closing the Program

```
// tree.cpp
void Tree::close() {
    write_tofile();
    emptytree(root);
}

void Tree::write_tofile() {
    if ( !( phonebook = fopen( filename, "w+" ) ) ) {
        cerr << "Could not open file" << endl;
        return;
    }
    write_inorder(root);
    fclose(phonebook);
}
```

Write In Order

```
// tree.cpp
void Tree::write_inorder(Phone_node *p) {
    struct File_Record {
        char name[NAME_LENGTH];
        char phonenum[PHONENUM_LENGTH];
    };
    File_Record record;
    if (p) {
        write_inorder(p->left);
        strcpy(record.name, p->name);
        strcpy(record.phonenum, p->phonenum);
        fwrite(&record, sizeof(File_Record), 1, phonebook);
        write_inorder(p->right);
    }
}
```

Question

- If we write the tree to the file in order, what kind of problem would arise?
- Answer:
 - The data in the tree will be written to the file in alphabetical order.
 - When reading from the file and recreating the tree, the tree will be unbalanced.
 - The performance of the search operation in an unbalanced tree in the worst case may become the same as that in a linked list.
 - That is why implementing a "write_preorder" function instead of a "write_inorder" function will be more meaningful.

write_preorder

```
// tree.cpp
void Tree::write_preorder(Phone_node *p) {
    struct File_Record {
        char name[NAME_LENGTH];
        char phonenum[PHONENUM_LENGTH];
    };
    File_Record record;
    if (p) {
        strcpy(record.name, p->name);
        strcpy(record.phonenum, p->phonenum);
        fwrite(&record, sizeof(File_Record), 1, phonebook);
        write_preorder(p->left);
        write_preorder(p->right);
    }
}
```

To Delete the Tree from Memory (Postorder Logic)

```
// tree.cpp
void Tree::emptytree(Phone_node *p) {
    if (p) {
        if (p->left != NULL) {
            emptytree(p->left);
            p->left = NULL;
        }
        if (p->right != NULL) {
            emptytree(p->right);
            p->right = NULL;
        }
        delete p;
    }
}
```

Searching for a Record

```
// phoneprog.cpp
void search_record() {
    char name[NAME_LENGTH];
    cout << "Please enter the name"
        << " of person you want to search for"
        << "(Press '*' for complete list):"
        << endl;
    cin.ignore(1000, '\n');
    cin.getline(name, NAME_LENGTH);
    if ( book.search(name) == 0 ) {
        cout << "Could not find record"
            << " matching search criteria" << endl;
    }
    getchar();
}
```

Searching for a Record

```
// tree.cpp
int Tree::search(char *search_name) {
    Phone_node *traverse;
    traverse = root;
    int countfound = 0;
    bool all = false;
    if (search_name[0] == '*')
        all = true;
    if (all) {
        traverse_inorder(root);
        countfound++;
    }
    else { // single record search
        while (traverse && !countfound) {
            int comparison =
                strcmp(search_name, traverse->name);
            if (comparison < 0)
                traverse = traverse->left;
            else if (comparison > 0)
                traverse = traverse->right;
            else { // if names are equal, record found
                cout << traverse->name << " "
                    << traverse->phonenum << endl;
                countfound++;
            }
        }
    }
    return countfound;
}
```

Traversing the Whole Tree

// tree.cpp

- For a printout of all data:

```
void Tree::traverse_inorder(Phone_node *p) {  
    if (p){  
        traverse_inorder(p->left);  
        cout << p->name << " " << p->phonenum << endl;  
        traverse_inorder(p->right);  
    }  
}
```

- At the end of this traversal, the names in the tree will be listed in alphabetical order.
- If we perform inorder traversal in the binary search tree, the data will be ordered from the smallest to the largest.

Adding a Record

```
// phoneprog.cpp
void add_record(){
    Phone_Record newrecord;
    cout << "Please enter the information for the"
        << "person you want to record" << endl;
    cout << "Name : ";
    cin.ignore(1000, '\n');
    cin.getline(newrecord.name, NAME_LENGTH);
    cout << "Phone number :";
    cin >> setw(PHONENUM_LENGTH)
        >>newrecord.phonenum;
    book.add(&newrecord);
    cout << "Record added" << endl;
    getchar();
}
```

Adding a Record: Adding a Node to the Tree

```
// tree.cpp
void Tree::add(Phone_node *toadd) {
    Phone_node *traverse, *newnode;
    traverse = root;
    int comparison;
    bool added = false;
    newnode = new Phone_node;
    strcpy(newnode->name, toadd->name);
    strcpy(newnode->phonenum,
           toadd->phonenum);
    newnode->left = NULL;
    newnode->right = NULL;
    if (root == NULL){
        //first node being added
        root = newnode;
        nodecount++;
        return;
    }
    while ((traverse != NULL) && (!added)){
        comparison = strcmp(newnode->name,
                             traverse->name);
        if (comparison < 0) {
            if (traverse->left != NULL)
                traverse = traverse->left;
            else {
                traverse->left = newnode;
                added = true;
            }
        }
        else if (comparison > 0) {
            if (traverse->right != NULL)
                traverse = traverse->right;
            else {
                traverse->right = newnode;
                added = true;
            }
        }
        else
            cout << "Data cannot repeat.\n";
    }
    if (added) nodecount++;
}
```

Updating a Record

// phoneprog.cpp

```
void update_record() {  
    char name[NAME_LENGTH];  
    char choice;  
    cout << "Please enter the name"  
        << " of the person you want"  
        << " to update:" << endl;  
    cin.ignore(1000, '\n');  
    cin.getline(name, NAME_LENGTH);  
  
    int personcount = book.search(name);  
    if (personcount == 0) {  
        cout << "Could not find"  
            << " record matching"  
            << " criteria" << endl;  
    }  
    else {  
        ...  
    }  
    getchar();  
}
```

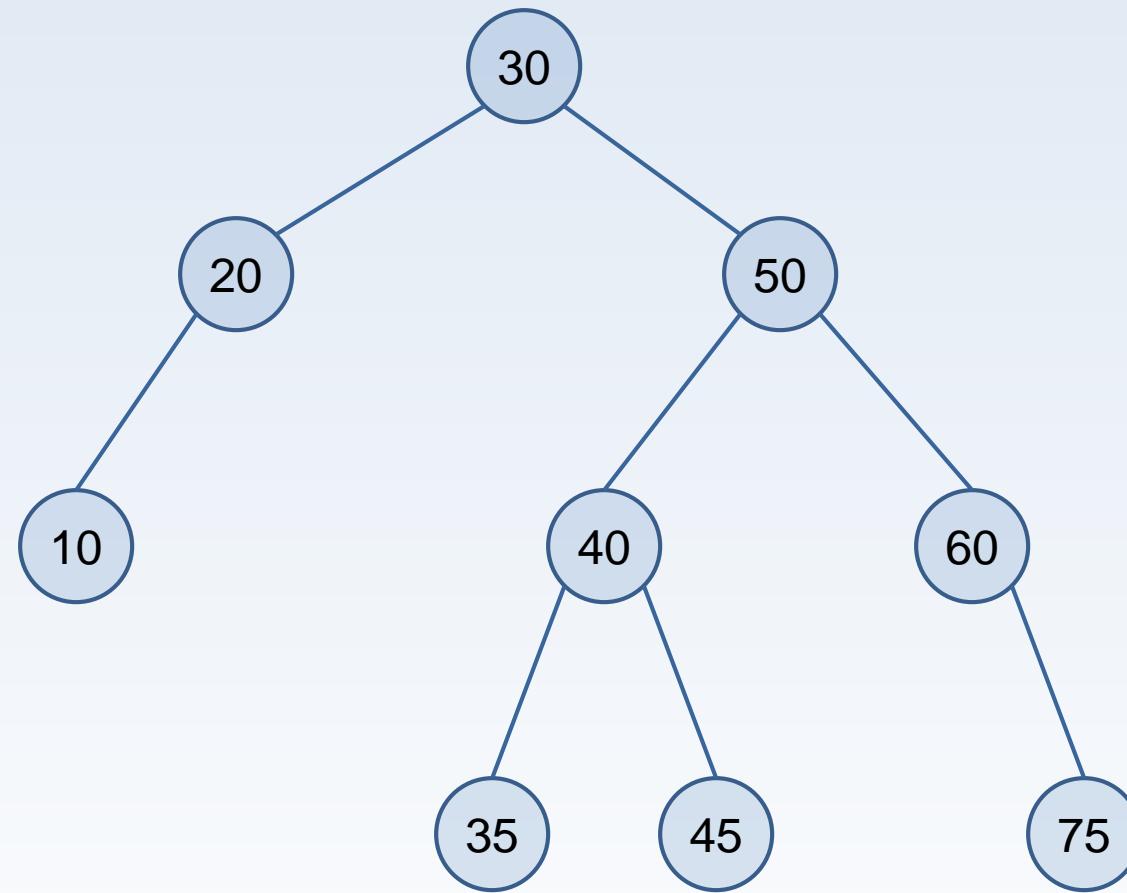


```
    cout << "Record found." << endl;  
    cout << "Do you want to update?"  
        << " (y/n) ";  
    do {  
        cin >> choice;  
    } while (choice != 'y' && choice != 'n');  
    if (choice == 'n') return;  
    Phone_Record newrecord;  
    cout << "Please enter current info"  
        << endl;  
    cout << "Name : " ;  
    cin.ignore(1000, '\n');  
    cin.getline(newrecord.name, NAME_LENGTH);  
    cout << "Phone number :";  
    cin >> setw(PHONENUM_LENGTH)  
        >> newrecord.phonenum;  
    book.remove(name);  
    book.add(&newrecord);  
    cout << "Record successfully updated"  
        << endl;
```

Removing a Record

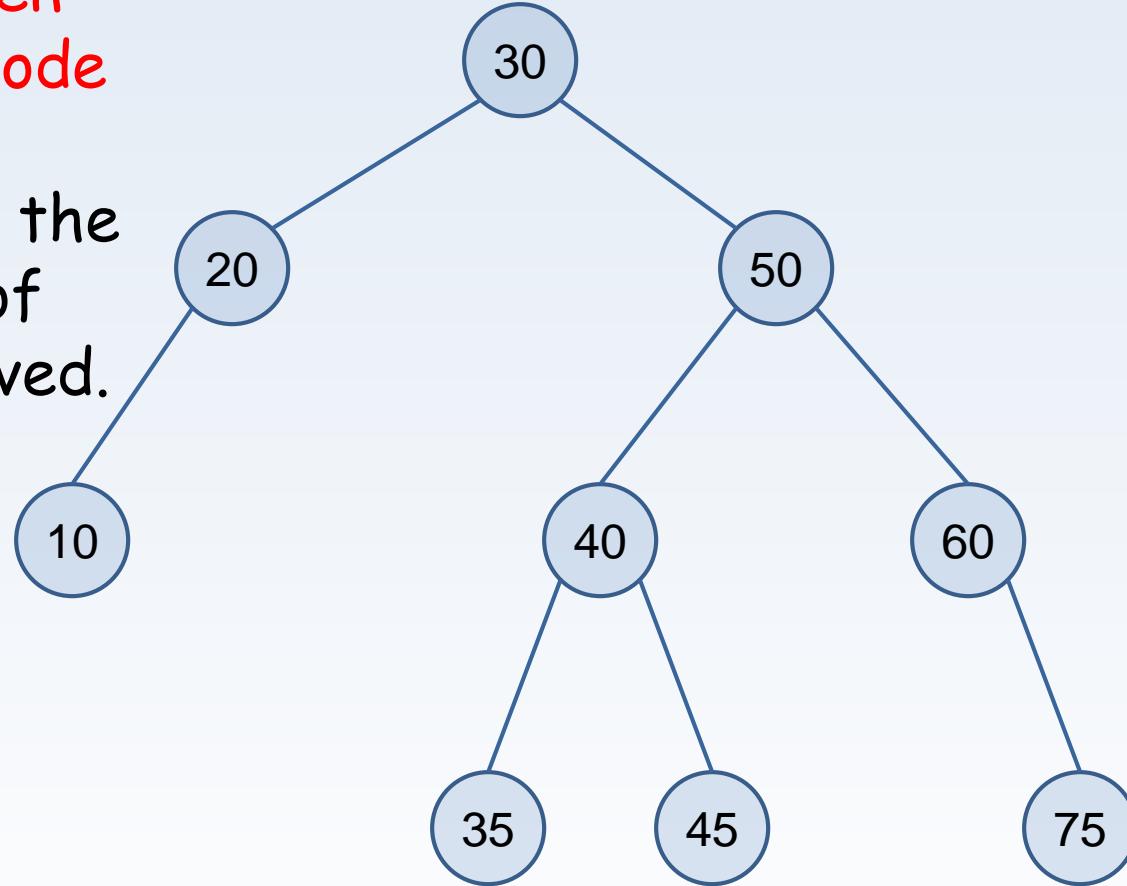
- The operation of removing from a binary search tree consists of two stages.
 - Searching for the element to be removed
 - Performing the remove operation
- Only the desired element should be removed from the tree without breaking the structure of the search tree.
- There are four cases to consider:
 - Removing a leaf
 - Removing a node that has only a left child
 - Removing a node that has only a right child
 - Removing a node that has both children

Removing a Leaf



Removing a Node that Has a Single Child

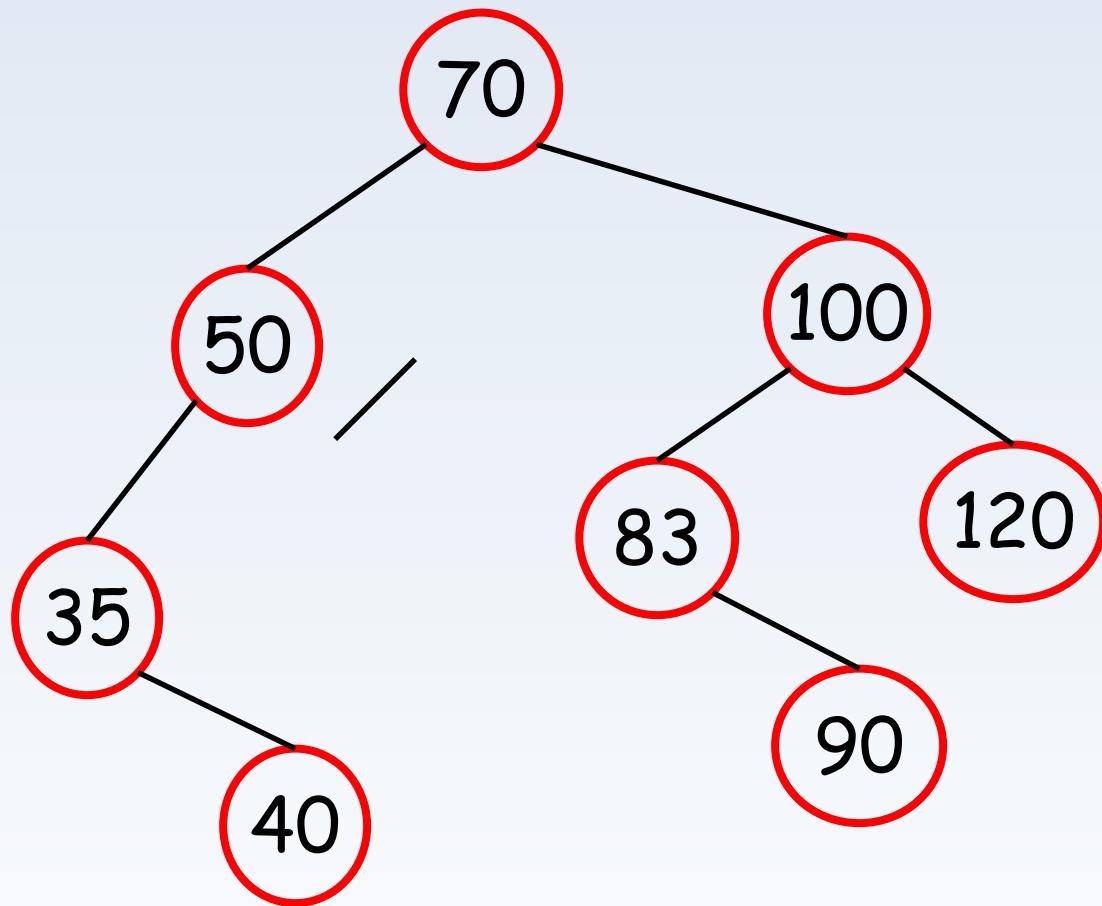
- If the node has a single child, **the link between the parent and the node to be removed is established between the parent and subtree of the node to be removed.**



Removing a Node That Has Both Children

- We move the right subtree to where the removed node used to be, and
- We try to link the left subtree to an appropriate node in the right subtree.
 - We must link the left subtree to the left of the leftmost child of the right subtree.

Removing a Node That Has Both Children: Example



Removing a Record

```
// phoneprog.cpp
void remove_record() {
    char name[NAME_LENGTH];
    char choice;
    cout << "Please enter name of person you want to delete:" << endl;
    cin.ignore(1000, '\n');
    cin.getline(name,NAME_LENGTH);
    int personcount = book.search(name);
    if (personcount == 0) {
        cout << "Could not find record matching search criteria" << endl;
    }
    else {
        cout << "Is this the record you want to delete?(y/n)";
        do {
            cin >> choice;
        } while (choice != 'y' && choice != 'n');
        if (choice == 'n') return;
        book.remove(name);
        cout << "Record removed" << endl;
    }
}
```

Removing a Record

// tree.cpp

```
void Tree::remove(char *remove_name){  
    Phone_node *traverse, *parent;  
    traverse = root;  
    bool found = false;  
    char direction = 'k';  
    while (traverse && !found) {  
        int comparison = strcmp(remove_name, traverse->name);  
        if (comparison < 0) {  
            parent = traverse;  
            direction = 'l';  
            traverse = traverse->left;        if (found) {  
                }  
        }  
        else if (comparison > 0) {  
            parent = traverse;  
            direction = 'r';  
            traverse = traverse->right;  
        }  
        else // found record to remove  
            found = true;  
    }  
    if (found) {  
        if (direction == 'l') {  
            parent->left = traverse->left;  
            parent->right = traverse->right;  
        }  
        else {  
            parent->left = traverse->left;  
            parent->right = traverse->right;  
        }  
        delete traverse;  
    }  
    else  
        cout << "Could not find"  
        << " record to remove.\n";  
}
```

Removing a Node

- There are four possible cases:
 - Removing a leaf
 - Removing a node that has only a left child
 - Removing a node that has only a right child
 - Removing a node that has both children

Removing a Leaf

```
// tree.cpp
if (traverse->left == NULL && traverse->right == NULL) {

    switch (direction) {
        case 'l':
            parent->left = NULL;
            break;
        case 'r':
            parent->right = NULL;
            break;
        default:
            root = NULL;
            break;
    }
}
delete traverse;
```

Removing a Node That Has Only Left Child

```
// tree.cpp
else if (traverse->right == NULL) {
    switch (direction) {
        case 'l':
            parent->left = traverse->left;
            break;
        case 'r':
            parent->right = traverse->left;
            break;
        default:
            root = traverse->left;
            break;
    }
}
delete traverse;
```

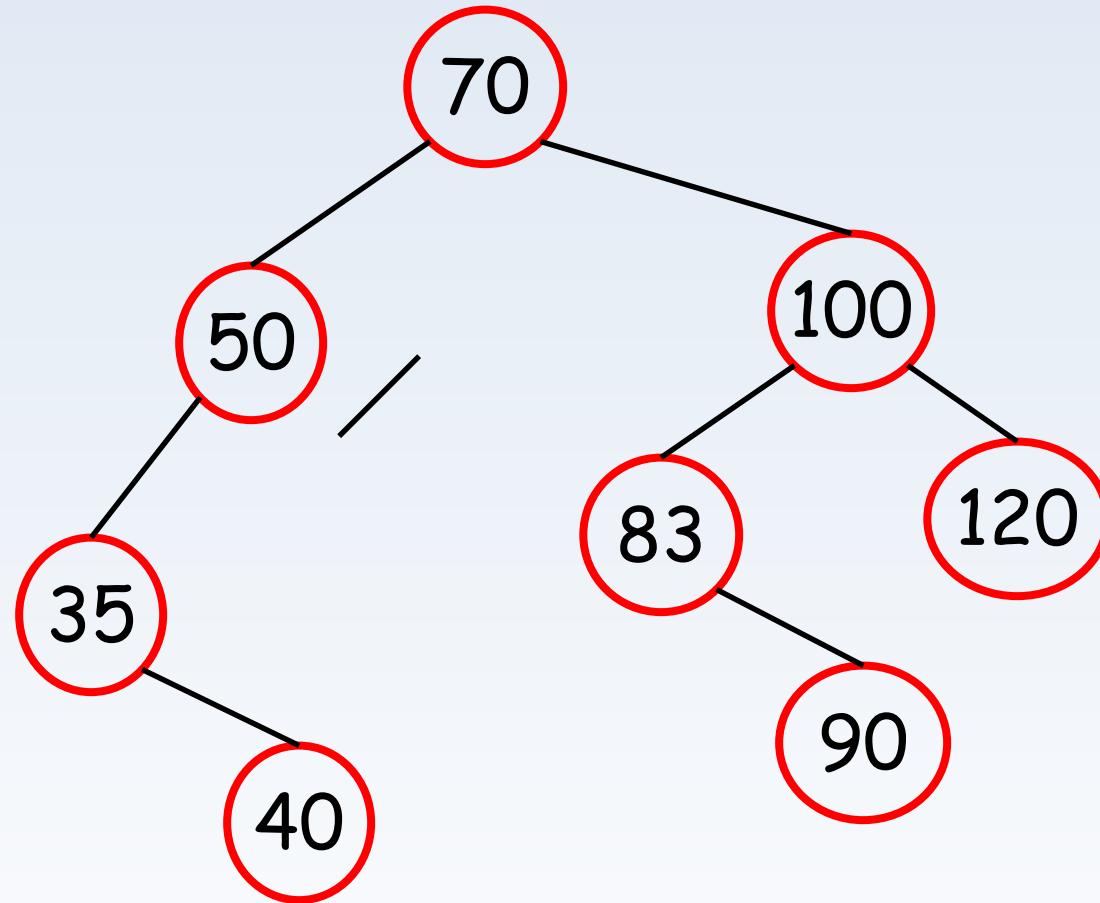
Removing a Node That Has Only Right Child

```
// tree.cpp
else if (traverse->left == NULL) {
    switch (direction) {
        case 'l':
            parent->left = traverse->right;
            break;
        case 'r':
            parent->right = traverse->right;
            break;
        default:
            root = traverse->right;
            break;
    }
}
delete traverse;
```

Removing a Node That Has Both Children

```
// tree.cpp
else {
    Phone_node *q = traverse->right;
    while ( q->left )
        q = q->left;
    q->left = traverse->left;
    switch (direction) {
        case 'l':
            parent->left = traverse->right;
            break;
        case 'r':
            parent->right = traverse->right;
            break;
        default:
            root = traverse->right;
            break;
    }
}
delete traverse;
```

Example



Removing a Record

// tree.cpp

```
void Tree::remove(char *remove_name) {  
    Phone_node *traverse, *parent;  
    traverse = root;  
    bool found = false;  
    char direction = 'k';  
    while (traverse && !found) {  
        int comparison = strcmp(remove_name, traverse->name);  
        if (comparison < 0) {  
            parent = traverse;  
            direction = 'l';  
            traverse = traverse->left;    if (found){  
        }  
        else if (comparison > 0) {  
            parent = traverse;  
            direction = 'r';  
            traverse = traverse->right;  
        }  
        else // found record to remove  
            found = true;  
    }  
    if (found){  
        if (traverse->left && traverse->right){  
            Phone_node *temp = traverse->right;  
            parent->right = temp;  
            temp->parent = parent;  
            temp->left = traverse->left;  
            temp->left->parent = temp;  
            temp->right = traverse->right;  
            temp->right->parent = temp;  
            delete traverse;  
        }  
        else if (traverse->left){  
            parent->right = traverse->left;  
            parent->right->parent = parent;  
            delete traverse;  
        }  
        else if (traverse->right){  
            parent->right = traverse->right;  
            parent->right->parent = parent;  
            delete traverse;  
        }  
        else  
            cout << "Could not find"  
            << " record to remove.\n";  
    }  
}
```

Removing a Record

// tree.cpp

```
void Tree::remove(char *remove_name) {
    Phone_node *traverse, *parent;
    traverse = root;
    bool found = false;
    char direction = 'k';
    while (traverse && !found) {
        int comparison = strcmp(remove_name, traverse->name);
        if (comparison < 0) {
            parent = traverse;
            direction = 'l';
            traverse = traverse->left;
        }
        else if (comparison > 0) {
            parent = traverse;
            direction = 'r';
            traverse = traverse->right;
        }
        else // found record to remove
            found = true;
    }
    if (found){
        if (direction == 'l')
            remove(&parent->left);
        else if(direction == 'r')
            remove(&parent->right);
        else remove(&root);
    }
    else
        cout << "Could not find"
            << " record to remove.\n";
}
```

Removing a Record

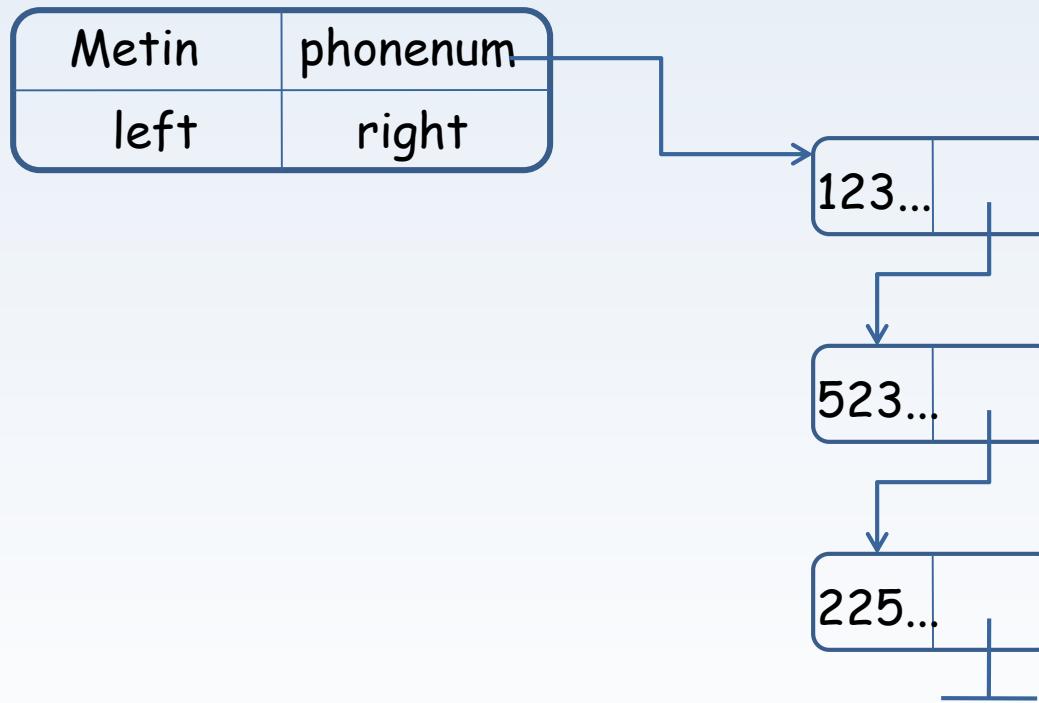
// tree.cpp

```
void Tree::remove(Phone_node **p) {  
    Phone_node *r, *q; // used to find place for left subtree  
    r = *p;  
    if (r == NULL) // attempt to delete nonexistent node  
        return;  
    else if (r->right == NULL) {  
        *p = r->left; // reattach left subtree  
        delete r;  
    }  
    else if (r->left == NULL) {  
        *p = r->right; // reattach right subtree  
        delete r;  
    }  
    else { // neither subtree is empty  
        for (q = r->right; q->left; q = q->left); // inorder successor  
        q->left = r->left; // reattach left subtree  
        *p = r->right; // reattach right subtree  
        delete r;  
    }  
}
```

p → pointer that will change
r → traverse

Practice

- If every person has more than one phone number
 - Add a linked list head pointer to the phonenum field of each node.



Data Structures

Ready Made
Data Structure Libraries

Ready Made Data Structures

- Today, many programming languages contain ready made data structures.
- Programmers can easily use these ready made data structures to solve the problems they want to solve independently of the content (code) of the data structures they will use.
- In C++, these ready made data structures are in the **Standard Template Library (STL)**.

STL

- The main idea of STL is making the complicated parts (coding) of data structures ready and presenting an easy-to-use interface to the programmer.
- For example, a programmer wanting to use a stack of integers can simply create this stack by writing the code below:

```
stack<int> s;
```

- Then, using the methods the STL provides for this data structure, the programmer can carry out the desired operations:

```
s.push(3);
```

Three Types of Data Structures in the STL

- **Sequence containers**
(linear data types such as vector and linked list)
 - C++ Vectors
 - C++ Lists
 - C++ Double-Ended Queues
- **Container adapters**
(versions of linear data structures with some properties restricted:
stack, queue)
 - C++ Stacks
 - C++ Queues
 - C++ Priority Queues
- **Associative containers**
(data structures that provide direct access to their elements via
search keys)
 - C++ Bitsets
 - C++ Maps
 - C++ Multimaps
 - C++ Sets
 - C++ Multisets

STL Libraries

- STL containers are located in different libraries.
- To use a container the related library should be added to the code:

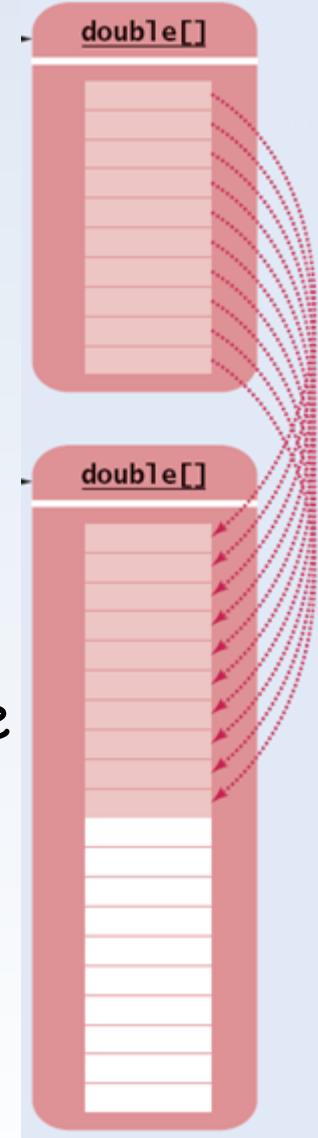
```
<vector>
<list>
<deque>
<queue>
<stack>
<map>
<set>
<valarray>
<bitset>
```

Sequence Containers

- In C++, there are 3 sequence containers:
 - `vector` - a more efficient array structure
 - `list` - linked list
 - `deque` - a type of array structure
- Common methods in sequence containers
 - `front` → returns the reference of the first element
 - `back` → returns the reference of the last element
 - `push_back` → adds an element to the end
 - `pop_back` → removes the last element

Vector

- Is a more reliable array realization.
- Similar to normal arrays, is made up of elements that are located consecutively in memory. The [] operator may be used. Fast access is to data is provided.
- Can dynamically increase its size:
 - When an array with a larger size is needed, it allocates a larger space from memory, and the content of the old array gets copied into this space.
 - The memory allocated from the memory for the old array can be returned to the memory.



Vector Example

```
vector<int> myvec;
int i;
for(i=0;i<5;i++)
    myvec.push_back(i);
for(i=0;i<5;i++)
    cout<<myvec[i]<<" ";
cout<<endl<<"Vector size="<<myvec.size()<<endl;
cout<<"Vector capacity="<<myvec.capacity()<<endl;
cout<<endl<<"Data to be added exceeds vector capacity"<<endl;
int exsize=myvec.size();
int excap=myvec.capacity();
for(i=exsize;i<2*excap;i++)
    myvec.push_back(i);
cout<<"Vector size="<<myvec.size()<<endl;
cout<<"Vector capacity="<<myvec.capacity()<<endl;
myvec.push_back(i+1);
cout<<endl<<"Data is being added to vector"<<endl;
cout<<"Vector size="<<myvec.size()<<endl;
cout<<"Vector capacity="<<myvec.capacity()<<endl;
while(!myvec.empty()){
    cout<<myvec.back()<<" ";
    myvec.pop_back();
}
```

Vector Example

```
0 1 2 3 4
```

```
Vektorun boyutu=5
```

```
Vektorun kapasitesi=8
```

```
Vektore kapasitesinden fazla veri ekleniyor
```

```
Vektorun boyutu=16
```

```
Vektorun kapasitesi=16
```

```
Vektore bir veri ekleniyor
```

```
Vektorun boyutu=17
```

```
Vektorun kapasitesi=32
```

```
17 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Pre:
```

Vector Member Functions

In addition to common methods, there are many other methods that can be used:

- Methods related to capacity: `size`, `max_size`, `resize`, `capacity`, `empty`, `reserve`
- Methods related to access to elements: `[]`, `at`, `front`, `back`
- Methods that make changes to the vector: `assign`, `push_back`, `pop_back`, `insert`, `erase`, `swap`, `clear`

You can refer to STL guides for usage details of methods belonging to containers.

List

- The STL container “list” presents an efficient implementation of adding to and removing from the linked list. If addition and removal operations are generally taking place at the end of the list, the “deque” container is more suitable.
- The “list” container is implemented as a doubly linked list. Therefore, we can traverse the list quickly in both directions.

Deque

- “double-ended queue”
- The deque is a structure that has been designed to use the best of the vector and list structures together.
- Like the vector, it provides fast access to elements using an index [] (this is implemented on the array.)
- As in the list, operations such as fast addition and removal of elements at the front and the back have been implemented in an efficient manner (But, it is also possible to insert in between elements of the list.)
- It is the default data structure for a queue structure.
- In addition to the basic operations in a vector, contains the `push_front` and `pop_front` functions.
- **Note:** `push_back` exists in all containers, but `push_front` exists in only list and deque.

Container Adapters

- In C++, there are 3 container adapters:
 - stack
 - queue
 - priority_queue

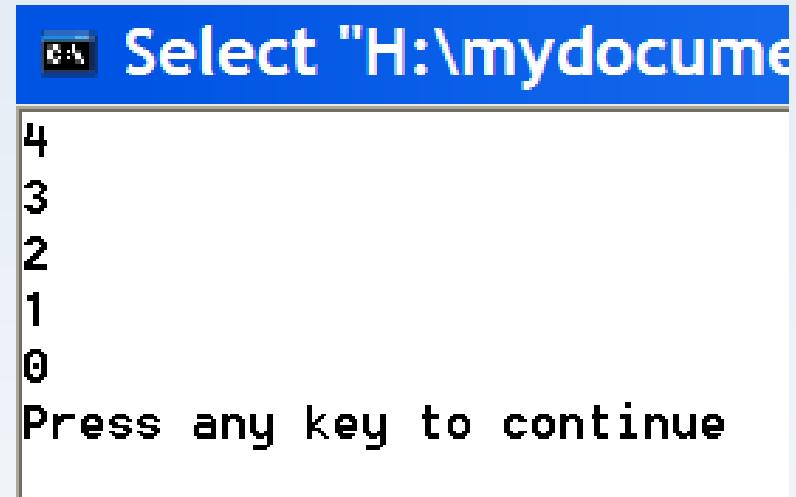
Stack Member Functions

- empty → isempty()
- pop → its difference from our pop() method is that it does not have a return value. void pop(); removes the element at the top of the stack but does not provide access to the element.
- push → push(...)
- size → returns the number of elements in the stack
- top → provides access to the topmost element of the stack but does not remove the element from the stack.

Stack Example

```
#include <iostream>
#include <stack>
using namespace std;

int main(){
    stack<int> stack;
    for(int i=0;i<5;i++)
        stack.push(i);
    while(!stack.empty()){
        cout<<stack.top()<<endl;
        stack.pop();
    }
    return EXIT_SUCCESS;
}
```



4
3
2
1
0
Press any key to continue

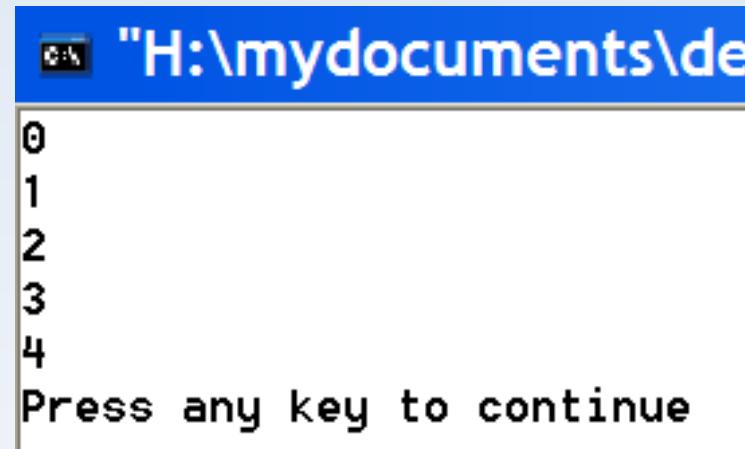
Queue Member Functions

- back → provides access to the last element of queue
- empty → isempty()
- front → provides access to the first element of queue
- pop → its difference from our pop() method is that it does not have a return value. void pop(); removes the first element of the queue but does not provide access to the element.
- push → push()
- size → returns the number of elements in the queue

Queue Example

```
#include <iostream>
#include <stack>
#include <queue>
using namespace std;

int main(){
    queue<int> myq;
    for(int i=0;i<5;i++)
        myq.push(i);
    while(!myq.empty()){
        cout<<myq.front()<<endl;
        myq.pop();
    }
    return EXIT_SUCCESS;
}
```



Priority Queue

- By default, it is ordered from largest to smallest.
- Largest gets processed first.
- It is also possible to specify your own priority type, but more advanced programming knowledge is needed.

empty

size

top → provides access to the first element

push → add element based on priority

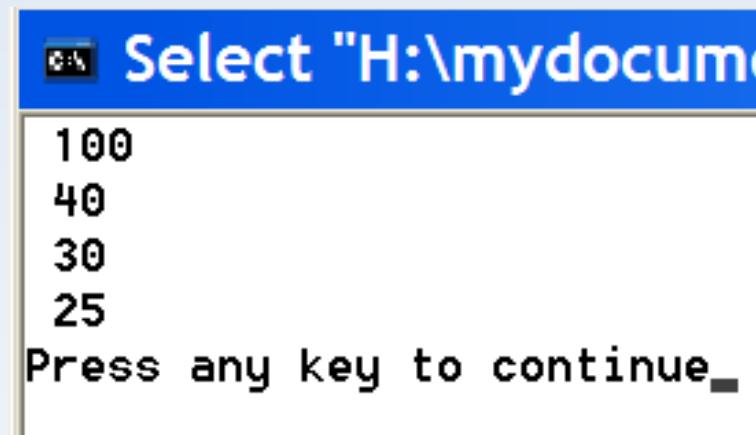
pop → removes the first element.

Priority Queue Example

```
#include <iostream>
#include <stack>
#include <queue>
using namespace std;

int main(){
    priority_queue<int> mypq;

    mypq.push(30);
    mypq.push(100);
    mypq.push(25);
    mypq.push(40);
    while (!mypq.empty())
    {
        cout << " " << mypq.top()<<endl;
        mypq.pop();
    }
    return EXIT_SUCCESS;
}
```



Container Adapters

- The advantage of a container adapter is that the user can choose the underlying data structure.
- For example, the underlying data structure for the stack structure is the deque by default. It has been implemented using the deque.
- If we want to use the version of the stack realized on the linked list:
 - `stack<int> stack;` is replaced with
 - `stack<int, list<int> > stack;`
- If we want to do this using a vector, we write
 - `stack<int, vector<int> > stack;`

Associative Containers

- Associative containers:
 - provide direct access to elements via keys
 - store search keys in order
- There are 5 associative containers:
 - multiset - holds only keys, repeats are possible
 - set - holds only keys, repeats are not possible
 - bitset - set used for bit operations
 - multimap - holds keys and associated values, repeats are possible.
 - map - holds keys and associated values, repeats are not possible.
- Common functions:
 - `find`, `lower_bound`, `upper_bound`, `count`

Example of Using Keys (map)

```
map<string, string> strMap;
strMap["Monday"]      = "1";
strMap["Tuesday"]     = "2";
strMap["Wednesday"]   = "3";
strMap["Thursday"]    = "4";
strMap["Friday"]      = "5";
strMap["Saturday"]    = "6";
strMap.insert(pair<string, string>("Sunday", "7"));
while(!strMap.empty()){
    cout<<strMap.begin()->first<< " "
        <<strMap.begin()->second<<endl;
    strMap.erase(strMap.begin());
}
```

Carsamba 3
Cuma 5
Cumartesi 6
Pazar 7
Pazartesi 1
Persembe 4
Sali 2

Press any key to continue

Iterators

- When we use the ready made libraries of C++ for data structures, we use structures called **iterators** to iterate over the elements of a data structure.
 - Iterators resemble pointers.
 - Just as we use pointers when traversing a linked list,
 - Ex: `Node *ptr=head;`
 - `while(ptr!=NULL) ptr=ptr->next;`
- when traversing the list structure of the STL, we use iterators. (We do not know what the node structure of the "list" is. We are only processing the data part.)

Iterators

```
list <int> myl;  
for(int i=0;i<5;i++)  
    myl.push_back(i);  
list <int>::iterator myit;  
for(myit=myl.begin();myit!=myl.end();myit++)  
    cout<<*myit<<endl;
```

Iterators

- It is not possible to use iterators on container adapters. This is because the way their elements can be accessed is predefined by the adapter structure.
- Ex: It is not possible to move between the elements of a stack. This is because access to the stack is possible only from a single point and this is implemented using the stack structure's own methods.

Iterator types	Direction	Capability
iterator	forward	read/write
const_iterator	forward	read
reverse_iterator	backward	read/write
reverse_const_iterator	backward	read

Iterators

```
list <int> myl;  
for(int i=0;i<5;i++)  
    myl.push_back(i);  
list <int>::reverse_iterator myit;  
for(myit=myl.rbegin();myit!=myl.rend();myit++)  
    cout<<*myit<<endl;
```

Example

To measure the frequency of words in a file containing English plaintext, we will write a program that uses the "map" data type.

Topic: STL and sequential access files

As output, the program will display the 10 most frequently used words in the file.

Example: Word Frequency

- First, words read in order from the file should be put into a map structure. In this way, the frequency of each word will be calculated.

```
FILE *myfile= fopen( "english.txt", "r" );
if(!myfile){ ... }
char word[100];
map <string,int> freq;
while(!feof(myfile)){
    fscanf(myfile,"%s",word);
    freq[word]++;
}
fclose(myfile);
```

Example: Word Frequency

- The map data structure is ordered only by key.
- It is not possible to order by value.
- For that reason, the map should be assigned to another map such that not the words, but the value fields are the keys.
- Since there will be more than one word with the same frequency value, the structure has to be a multimap:

```
multimap <int, string > freq_rev;  
map<string, int>::iterator it;  
for(it=freq.begin(); it!=freq.end(); it++)  
    freq_rev.insert(make_pair(it->second, it->first));
```

Example: Word Frequency

- In the new multimap, values are ordered from smallest to largest.
- To print the most frequent 10 words to the screen, the 10 records at the end of the structure have to be found.

```
multimap <int, string>::reverse_iterator myit;  
int count;  
for (myit=freq_rev.rbegin(), count=0; count<10; myit++, count++)  
    cout<<(*myit).second<< " "<<(*myit).first<<endl;
```

Example: Word Frequency

the	145
of	97
and	73
a	64
to	56
writing	44
is	43
in	42
as	35
was	24