UNIT 3
Concrete Data Types

- Classification of Data Structures
- Concrete vs. Abstract Data Structures
- Most Important Concrete Data Structures
  - Arrays
  - Records
  - Linked Lists
  - Binary Trees

Overview of Data Structures

- There are two kinds of data types:
  - *simple* or *atomic*
  - *structured data types* or *data structures*
- An *atomic* data type represents a single data item.
- A *data structure*, on the other hand, has
  - a number of components
  - a structure
  - a set of operations
- Next slide shows a classification of the most important data structures (according to some specific properties)
Concrete Versus Abstract Types

- **Concrete data types** or **structures** (CDT's) are direct implementations of a relatively simple concept.
- **Abstract Data Types** (ADT's) offer a high level view (and use) of a concept independent of its implementation.
- Example: Implementing a student record:
  - CDT: Use a struct with public data and no functions to represent the record
    - does not hide anything
  - ADT: Use a class with private data and public functions to represent the record
    - hides structure of the data, etc.
- Concrete data structures are divided into:
  - contiguous
  - linked
  - hybrid
- Some fundamental concrete data structures:
  - arrays
  - records,
  - linked lists
  - trees
  - graphs.
C++ Arrays

- A C++ array has:
  - a collection of objects of the same type
  - a set of index values in the range [0, n]

- **Structure:**
  - objects are stored in consecutive locations
  - each object has a unique index

- **Operations:**
  - \[i\] accesses the \((i+1)\)th object

E.g. In

```cpp
char word[8];
```

- word is declared to be an array of 8 characters
- 8 is the *dimension* of the array
- dimension must be known at compile time.

C++ Arrays (cont’d)

- **Array indices (or subscripts)** start at 0.
- word 's elements are:
  - word[0], word[1], ..., word[7]
- An array can also be initialized, but with constants only
  - `int ia[] = {1,2,0};`

- Arrays cannot be assigned to one another; each element must be assigned in turn
Arrays & Pointers

- Are closely related.
The declaration:
  ```
  type a[10]
  ```
  - allocates space for 10 items of type type
  - items are stored in consecutive memory locations

- C++ treats consecutive elements in the array having consecutive addresses:
  ```
  &a[0] < &a[1] < ... < &a[9]
  and
  &a[1] = &a[0] + 1
  &a[i] = &a[i-1] + 1
  ```

- `a` is a variable of type pointer to `type`,
- `&a[i]` is the same as `a+i
- `a[i]` is the same as `*(a+i)

There are two ways to access the elements of an array. We can use either:
- array subscripts, or
- pointers

Example

- Suppose we declare:
  ```
  int a[10];
  ```

- Suppose we declare:
  ```
  int i, a[10]
  ```

The following two statements output the elements of a:
1. for (i = 0; i < 10; i++)
   ```
   cout << a[i];           // using subscripts
   ```
2. for (i = 0; i < 10; i++)
   ```
   cout << *(a + i);                // using pointers
   ```

- If
  ```
  int * p;
  ```
  then
  ```
  p = &a[i] or
  p = a + i
  ```
  makes `p` point to the i-th element of `a`.

- Straying beyond the range of an array results in a `segmentation fault`.
- Pointers are not integers
  - Exception: NULL (which is 0) can be assigned to a pointer.
  - NULL is the undefined pointer value

Pointers are not integers
  - Exception: NULL (which is 0) can be assigned to a pointer.
  - NULL is the undefined pointer value
Dynamic arrays

- Are declared as pointers
- Space for these arrays is allocated later, when the size is known

**Example:** Consider the declarations:

```c
int b[10];
int * a;
a = new int[10];
```

Then:
- `a` and `b` are both arrays
- `b` is a *fixed array*; `a` is a *dynamic array*
- `[]` can also be used on `a`
- `a[2]` is the third element of `a`

**BUT**
- `b` has space for ten integers
- `a` has space for one pointer
  - space for its elements must be allocated by `new`
- `b` is a constant pointer; `a` can be changed

- A dynamic array can be expanded
  - i.e., to expand `a` we can write:
    ```c
    int* temp = new int[10 + n];
    for (int i = 0; i<10; i++)
      temp[i] = a[i];
    delete a;
    a = temp;
    ```

**Example Using Dynamic Arrays:**

**Implementation of EmployeeDB using dynamic arrays:**

```
EmployeeDB (Dynamic Array)
```

---

Passing Array Parameters

- Arrays are always passed by reference
- Suppose we declare,
  ```c
  int a[10];
  ```

  To pass array `a` to a function `f`, `f` may be declared as:

  ```c
  type f( int d[], int size ) or
type f( int* d , int size )
  ```

- In any case, `f` is called by `f(a, sizeof a)`.
Multi-dimensional Arrays

- To store the temperature measured at each hour of each day for a week, we can declare:
  
  ```c
  int temp[7][24];
  ```

  - `temp` is a 2-dimensional array.
  - To get the temperature at noon on Monday, we can do:
    ```c
    temp[1][11];
    ```

- To pass a multi-dimensional array to a function, the first subscript can be free.

- i.e. To pass `temp` to `f`, `f` may be declared as
  ```c
  ... f(int t[][24], int size1); or
  ... f(int (*t)[24], int size1); or
  ... f(int** t, int size1, int size2)
  ```

- In the last declaration, `t` is a pointer to a pointer, while in the other two `t` is a pointer to an array of 24 integers.

Multi-dimensional Arrays vs. Pointers

- Given the declarations:
  ```c
  int t[7][24];
  int* s[7];
  int** r;
  ```

  - we can allocate adequate space to `s` and `r` so that `t`, `s` and `r` behave as 2d arrays

- But
  - `t` is a true 2-dimensional array (has space allocated)
  - `s` is an array of 7 pointers (each pointing to an array of 24 integers)
  - `r` is a pointer to a pointer

- A use of semi-dynamic arrays:
  - arrays with different row length

- i.e.
  ```c
  char* day[8] = {"Illegal day name", "Sunday", ..., "Friday"}
  ```
Features of Arrays

- Simple structures.
- Their size is fixed;
  - dynamic arrays can be expanded, but expansion is expensive.
- Insertion and deletion in the middle is difficult.
- Algorithms are simple.
- Accessing the i-th element is very efficient.

C++ Records (struct's)

- Records allow us to group data and use them together as a unit.
- The record type has:
  - a collection of objects of same or different type
  - each object has a unique name
  - .obname accesses the object with name obname.
- C++ uses "struct" for records. They are also called "structures" in C++.
- For instance, after declaring
  ```
  struct date {
      int day;
      char* month;
      int year;
  }
  ```
  date is now a new type; it can be used as:
  ```
  date today = {20, "jun", 1993};
  ```
- We can access the components of a structure using the select member operator ".".
  E.g.
  ```
  today.month[2] // 'n'
  ```
- A C++ struct may also have function members.
- The difference between classes and records:
  - by default, a class components are private,
  - while a struct's components are public.
C++ Records (cont’)

 Structures are commonly used to implement lists, trees, etc. An item of these types of structures usually looks like:

```cpp
struct item {
  int data;
  item* next;
};
```

We can then declare:

```cpp
item item1, item2, *head, *current;
```

The physical structure of this would look like the following:

```
head -> current
```

The operator `->`

 Structure components can be accessed by pointers using the `point at member` operator `"->"`.

 E.g. If we set

```cpp
head = &item1
```

then

```cpp
head -> data
```

is the data field of item1.

 Structures can be copied member-wise:

```cpp
item2 = *head
```

We can also pass a structure as a parameter to a function. However, it is usually more efficient to pass a structure by reference, or to pass a pointer to the structure instead.

 i.e.

```cpp
void f(const date& d ) or
void f(const date* d )
```
Linked Lists

- A (singly) linked list is a sequence of nodes linked together:

They represent sequences of data items.
- Each node in the list contains the item and a pointer to the next element.
- The last pointer is set to 0 (or NULL) to denote the end of the list.
- The whole list is defined by a pointer to the first item (called list here).
- In C++ the node and the list are defined as:

```cpp
// TYPE is the type of our items
typedef int TYPE;
struct node {
    TYPE item;
    node* next;
};
```

Or:

```cpp
typedef int TYPE;
class node {
    public:
        TYPE item;
        node* next;
};
```

Common Operations on Linked List

- **Insert an item in the list.** Many types of insertion:
  - `insert_first`: insert item at the front of list
  - `insert_last`: insert item at the end of the list
  - `insert_after`: insert item in list after a certain node
- **find**: finds the node in the list with a given item
- **delete_item**: removes an item from the list
- **printNode**: prints the contents of a node

- **A Singly Linked List Toolkit**
  The following files contain an implementation of a module (or toolkit) for the singly linked list structure:
  - [Singly Linked List](#)

- **Example Using Linked Lists**
  Implementation of EmployeeDB using singly linked lists:
  - [EmployeeDB (Linked List)](#)
Head Nodes

- Processing of this first node is different from processing of the other nodes.
- A **head node** is a dummy node at the beginning of the list.
  - It is similar to the other nodes, except that it has a special value
  - It is never deleted.
  - Processing every actual node is the same.
- Usually, it is more confusing and it is not used.

Circular Linked Lists (or rings)

- A circular linked list looks like:

```
head
```

```
 item1 -> item2 -> item3 -> item4 -> head
```

- A circular linked list is appropriate when there is no distinct first and last item.
- The algorithms for circular linked lists are similar to those for singly linked lists, except that
  - none of the links is null
  - the end of the list is reached when `curr->next == head`
Doubly-linked Lists

- Similar to singly linked lists except that each node also has a pointer to the previous node.
- Doubly linked list node definition:

  ```c
  struct dnode {
    TYPE item;
    dnode* next;
    dnode* prev;
  };
  ```

- Operations are defined similarly

A Doubly Linked List Toolkit:
Can be found in Doubly Linked List

Features of Linked Lists

Compared to arrays, linked lists have the following advantages/disadvantages:

- **Advantages**
  - Are dynamic structures; space is allocated as required.
  - Their size is not fixed; it grows as needed.
  - Insertion and deletion in the middle is easy.

- **Disadvantages**
  - More space is needed for the links.
  - Algorithms are more complex.
  - Impossible to directly access a node of the list.
Binary Trees

- A binary tree is a structure that
  - is either empty, or
  - it consists of a node called a root and two binary trees called the left subtree and the right subtree.
- Pictorially a binary tree looks like the following:

```
    A
   /|
  /  |\n B   C
 /   /|
D   E  F
    /|
     G
```

Parents, Children & Paths

- **Parents & Children:**
  - If there is a link from node N to M then N is the parent of M and M is a child of N.
  - The root has no parent.
  - A leaf is a node on the bottom level of the tree without any children.
  - A node can have a maximum of 2 children.
  - A tree cannot have cycles.
- **Grandparents, grand children, ancestors, descendants** are defined similarly.
- **Path from N1 to Nk**
  - a sequence of nodes N1, N2, ..., Nk, where Ni is a parent of Ni+1.
  - *path length*: # of nodes in the path from N1 to Nk (some authors use # of edges).
**Depth or level** of a node $N$
- length of the unique path from the root to $N$
- the level of the root is 1.

**Height of a node $N$:**
- length of the longest path from $N$ to a leaf
- a leaf's height is 1.

**Height of the tree:**
- height of its root

The number of nodes in a binary tree of height $h$ is $\geq h$ and $\leq 2^h - 1$ nodes.

---

**Implementation of Trees**

**Implementation of a binary tree in C++:**
- a node in the tree contains the item and two pointers to the subtrees:

```c
typedef int TYPE;
struct bnode {
    TYPE item;
    bnode* left;
    bnode* right;
};
```

- A C++ binary search tree is just a pointer to the root.
Common Operations for Binary Trees

- **Insert an item in the tree**: To the left or right of a node:
  - `insert_left`: insert item on the left of a given node
  - `insert_right`: insert item on the right of a given node
- **find**: finds the node in the tree with a given item
- **find_parent**: finds the parent of a given node in the tree
- **delete_node**: removes the node with the given item from the tree
- **print**: prints the whole tree (sideways)

A Binary Tree Toolkit

- An implementation of a module (or toolkit) for the binary tree structure can be found in the Examples:
  - Binary Tree

Traversing a binary tree

- There are three types of traversal.
  - `preorder`: node then left subtree then right subtree
  - `inorder`: left subtree then node then right subtree
  - `postorder`: left subtree then right subtree then node

- **Inorder traversal**: The following code applies a function `visit` to every node in the tree inorder:

```c
void inorder( bnode* root ) {
    // apply the function visit to every node in the tree, inorder
    if( root != NULL ) {
        inorder( root->left);
        visit ( root );   // apply visit to the root of the tree
        inorder( root->right);
    }
}
```

- Tree traversal is not usually implemented by a function. What is shown here is just an example.
Higher trees and Graphs

**N-ary Trees**
- Like binary trees except that a node may have up to $n$ subtrees.

**Graphs**
- More general than trees. They can have cycles and are usually hybrid structures.