Chapter 10
Data Structures

Roughly speaking, data structures refer to the ways data are organized. When there is no relation at all, we have sets. When linearly ordered, i.e., for any data, there is a unique predecessor and a unique successor, we have a linear structure. When we lift the restriction on the successor end, we have a tree structure; and if we lift both, we have the most general structure of a graph.

We already saw the tree structure when discussing heaps, and will see more of its application later. In this unit, we will focus on the linear structure, or simply a list.
(Dynamic) Sets

Sets, namely, collection of elements, are basic to computer science, as well as to mathematics. A difference is that when we work with sets in computer science related work, sets change, i.e., they can grow, shrink, or be modified, thus *dynamic*.

We will now discuss various ways to represent and manipulate such dynamic sets.
What do we represent?

In a typical implementation of a dynamic set, each element is represented with an object to which we often have a *pointer*, so that we can get to the *next* element. In other words, when being implemented, we do add in connection to the supposedly isolated elements, for a practical reason.

The components of these objects can be examined and manipulated, with various *methods*.

These objects contain an identifying *key* field, often taken from a totally ordered set of values, together with other *satellite data*, where the real data are kept.
The basic structure

Consider the following object, which contains two pieces: one to contain some information, and the other is really a reference to an object of the same type.

```java
class Node{
    int info; Node next;
}
```

Two objects of this class can be instantiated and linked together by letting the `next` piece, namely the pointer, of the first object become a reference to the second.

such an idea can certainly be generalized, and the `next` field of the last such object contains a `null`, indicating that it is indeed the last object in this list.
The following code adds a bunch of magazines into a rack, implemented as a dynamic list.

```java
public class MagazineRack{
    public static void main (String[] args){
        MagazineList rack = new MagazineList();

        rack.add(new Magazine("Time"));
        rack.add(new Magazine("Woodworking Today"));
        rack.add(new Magazine("Communications of the ACM"));
        rack.add(new Magazine("House and Garden"));
        rack.add(new Magazine("GQ"));

        System.out.println (rack);
    }
}

public class Magazine{
    private String title;

    public Magazine (String newTitle){
        title = newTitle;
    }
    public String toString (){ return title;
}
```
The MagazineList class

The following code shows a method for the MagazineList class that adds in a magazine as the last node in the list.

public class MagazineList{
    private MagazineNode list;
    MagazineList(){
        list = null;
    }

    public void add (Magazine mag){
        MagazineNode node = new MagazineNode (mag);
        MagazineNode current;
        if (list == null) list = node;
        else {
            current = list;
            while (current.next != null)
                current = current.next;
            current.next = node; }
    }

    Notice that in this example, we always add, or insert, a new node as the last one in the list.
The `MagazineNode` class

The following code shows a more specific node class, that of the `MagazineNode`, which is to contain a magazine.

```java
private class MagazineNode {
    public Magazine magazine;
    public MagazineNode next;

    //-----------------------------------
    // Sets up the node
    //-----------------------------------
    public MagazineNode (Magazine mag) {
        magazine = mag;
        next = null;
    }
}
```

Technically, it is an inner class of the above `MagazineList` class.
Some of the operations

Search(S, k), given a set and a key value k, returns a pointer to an object x with k as its key value.

Insert(S, x) modifies S by adding in an object x. We often assume that all the components of x are initialized.

Delete(S, x) also modifies S by removing an object x.

Minimum(S) returns an object with the smallest key value. Maximum(S) is similar.

Successor(S, x), where S is totally ordered, returns an object whose key value is next to the one as contained in x, and returns NIL, is x contains the maximum key value. Predecessor(S, x) behaves similarly.
Stacks and queues

Stacks and queues are examples of elementary data structures, where we work at some rather restricted position(s).

In a stack, the element to be deleted is always the most recently inserted; while in a queue, the element to be deleted is always the one that has stayed there for the longest time. Thus, a stack implements the \textit{last-in, first-out} (LIFO) policy, and a queue the \textit{first-in, first-out} (FIFO) policy.
Examples

There are plenty of examples for the stack structure, e.g., the following coin dispenser.

Stack also plays a critical role in implementing control transfer.

Queue is also widely used in both our daily life and computing fields.
import java.util.Stack;
import cs1.Keyboard;

public class Decode {
    public static void main (String[] args){
        Stack word = new Stack();
        String message; int index = 0;

        System.out.println ("Enter the coded message:");
        message = Keyboard.readString();
        System.out.println ("The decoded message is:");

        while (index < message.length()){
            while (index < message.length()
                && message.charAt(index) != ' ')
                word.push(new Character(message.charAt(index)));
            index++;
        }
        while (!word.empty())
            System.out.print (((Character)word.pop()).charValue());
        System.out.print (" "); index++;
    }

    System.out.println();
}
}
This example shows that we *push* a list of items, of type *Object*, into a stack, defined in *java.util*, and *pop* them out.

Notice that Java 2 Version 1.4 does not support *Queue*, but Version 1.5 does.

**Homework:** Figure out the Queue type as supported in Java, and convert the above decoder with a Queue, which prints out the stuff in the same order that were entered.
Implementing a stack

It is pretty easy to implement a stack containing at most $n$ elements by using an array $S[1..n]$, together with another piece of data $\text{top}[S]$, which indicates the top element of the stack. In particular, when $\text{top}[S]=0$, the stack is empty, and when $\text{top}[S]=n$, it is full.

When we try to pop an element when the stack is empty, it underflows, and will overflow when we try to push another into a full stack.

It is clear that the order in which the elements are pushed in will be the opposite of being popped out.
The code

STACK-EMPTY(S)
1. if top[S]=0
2. then return true
3. else return false

PUSH(S, x)
1. if STACK-FULL(S)
2. then error "overflow"
3. else top[S]<-top[S]+1
4. S[top[S]]<-x

POP(S, x)
1. if STACK-EMPTY(S)
2. then error "underflow"
3. else top[S]<-top[S]-1
4. return S[top[S]+1]

It is also easy to implement STACK-FULL(S). They all take $\Theta(1)$. 
The insertion operation on a queue is often called `Enqueue`, and the deletion `Dequeue`. Java uses `offer` and `poll` instead.

A queue, when implemented with an array, has a `head` and a `tail`. Whenever we want to enqueue an element, it is added at the tail, and whenever we want to dequeue, an element is taken from the head, just like what we do for any waiting line.

Technically, we implement a queue with at most $n - 1$ elements with an array $Q[1..n]$, together with two additional pieces $\text{head}[Q]$ and $\text{tail}[Q]$, which point to the head and the tail of the queue, respectively.
A few tricky details

1. While $\text{head}[Q]$ points to the next element to be dequeued, $\text{tail}[Q]$ points to the position where the next element will be added. Thus, the current queue contains the elements $\text{head}[Q]$, $\text{head}[Q]+1$, ..., $\text{tail}[Q]-1$.

2. To be space efficient, the queue is also wrapped around the array in the sense that the position next to the last position of the array will be the first one.

Thus, the array becomes a *circular one*.

3. When $\text{head}[Q]=\text{tail}[Q]$, the queue is empty, and when $\text{head}[Q]=\text{tail}[Q]+1$, the queue is full.
The code

ENQUEUE(Q, x)
1. Q[tail[Q]]<-x
2. if tail[Q]=length[Q]
   3. then tail[Q]<-1
   4. else tail[Q]<-tail[Q]+1

DEQUEUE(Q)
1. x<-Q[head[Q]]
2. if head[Q]=length[Q]
   3. then head[Q]<-1
   4. else head[Q]<-head[Q]+1
5. return x

They both take $\Theta(1)$.

**Homework:** Exercises 10.1-1(*), 10.1-3(*), and 10.1-5.
Linked lists

A linked list is a data structure in which the objects are put in a linear order, just like an array. In an array, the order is determined by the array indices. This is possible since array is a static structure in the sense that the whole array is allocated as a single block.

With a linked list, the space for the objects are allocated at different times, thus, we have to “record” the position of the next object as part of the current object in order to implement the successor operation. Hence, the position of an object in a linked list is provided by a pointer.

Linked list offers a simple, efficient and flexible implementation for the dynamic sets.
A linked list can be either *singly linked*, or *doubly linked*. Each element in a singly linked list has one pointer field, \( \text{next} \), besides the key value. Thus, holding on any element, it is possible to go to the next element, except in the case of the last one, whose next field contains NIL.

Each element in a doubly linked list contains two pointer fields, \( \text{next} \) and \( \text{prev} \). Besides going to the next, we can also go to the previous one directly, at the cost of an additional field.

A linked list can also be *circular*, where the next field of the last element points to the first, and the prev of the first points to the last.

It can also be *sorted*, then the order of the elements is the same as their key values.
Doubly linked list in Java

Below specifies the node, which has two pointer fields, one pointing to the successor and the other to the predecessor.

```java
public class DoubleNode {
    int info;
    DoubleNode prev;
    DoubleNode next;

    // When constructing a double node with key, 
    // set both pointers NIL
    public DoubleNode(int key)
        info=key;
        prev=next=null;
}
```
The doubly linked list itself can be defined as follows:

```java
public class DoubleLinkedList{
    private DoubleNode head;

    //Initialize an empty list
    public DoubleLinkedList(){
        head=null;
    }

    public DoubleNode search(int key);
    public void addFirst(int key);
    public void addLast(int key);
    public int delete(int key);
    private DoubleNode delete(DoubleNode node);
}
```

We will now fill in the details for those methods.
Let’s check out the details of implementing some of the operations in a doubly linked list.

The procedure list-Search(L, k) finds the first element with key k in L by a sequential search, and returns a pointer to that element. If nothing is found, it sends back NIL.

LIST-SEARCH(L, k)
1. x<-head[L]
2. while x != NIL and key[x]!=k
3. do x<-next[x]
4. return x

It is clear that this procedure takes \( \Theta(n) \) in the worst case.
The implementation of this search function in Java is as follows. Notice that we need to have this.head to indicate the list which it heads.

```java
public DoubleNode search(int key){
    //Set the first node to be current
    DoubleNode current=this.head;

    //When there is something, but it is
    //not the stuff, keep on looking
    //until the very end.
    while(current!=null && current.info!=key)
        current=current.next;

    return current;
}
```

Obviously, its runtime is $O(n)$, where $n$ is the length of the list.

**Question:** Does it work for an empty list?
Insert into a list...

Given an element \( x \) with its key field filled, the List-Insert procedure adds \( x \) at the \textit{beginning} of a list.

\begin{verbatim}
LIST-INSERT(L, x)
1. next[x]<-head[L]
2. if head[L]!=NIL
3. then prev[head[L]]<-x
4. head[L]<-x
5. prev[x]<-NIL
\end{verbatim}

It is clear that it takes \( \Theta(1) \) to apply this procedure.

\textbf{Question:} What happens if the list is empty?

\textbf{Homework:} Exercises 10.2-1, 10.2-3 and 10.2-7(*).
I want to see...

When working with linked lists, the most important thing is to know where a pointer is pointing at.

A figure often helps... .

The addLast is quite similar... .
...in java

The following method adds another node with a given key \textit{at the end} of the list:

```java
public void addLast(int key){
    DoubleNode node=new DoubleNode(key);
    DoubleNode current;

    if(head==null)
        this.head=node;
    else {
        current=this.head;
        //Looking for the last one
        while(current.next!=null)
            current=current.next;
        current.next=node;
        node.prev=current;
    }
}
```

Obviously, its runtime is $\Theta(n)$, where $n$ is the length of the list. \texttt{addFirst} can be similarly implemented.
Delete from a list

Delete(L, x) removes an element x from L. Given a pointer to x, this procedure takes it out by modifying all the related pointers.

LIST-DELETE(L, x)
1. if prev[x]!=NIL
2. then next[prev[x]]<-next[x]
3. else head[L]<-next[x]
4. if next[x]!=NIL
5. then prev[next[x]]<-prev[x]

It takes only $\Theta(1)$, and will take $\Theta(n)$ if given a key instead, where a search has to be done first.
The following deletes a node, referenced by node, from a doubly linked list.

```java
public DoubleNode delete(DoubleNode node) {
    //if the node to be deleted is the very first one simply make the next one to be the first; otherwise we have to let the previous one hook up to the next one.
    if (node.prev != null)
        (node.prev).next = node.next;
    else this.head = node.next;
    //If the node to be deleted is not the last one, we have to let the next node to be the next node of the previous node.
    if (node.next != null)
        (node.next).prev = node.prev;

    return node;
}
```
Another scenario could be that we are given a key value, instead of the pointer. Then, we can use \texttt{Search(Page, 23)} to find the pointer pointing to the element containing that key.

```java
public int delete(int key){
    DoubleNode current, sent;

    // Look for the one to be deleted
    current = search(key);
    // Found it!
    if(current != null){
        sent = delete(current);
        return sent.info
    }
    else return -1;
}
```

The runtime is $O(n)$. This is why we use a doubly linked list.

**Question:** Have you checked the project page recently?
Some of the programming languages, such as C and C++, provide the pointer data types. For example, a pointer to an integer can be declared as follows:

```c
int * y;
```

When the program needs to actually use the variable, we have to use `new` to ask for the space:

```c
y = new int;
```

Then, we can manipulate `*y`, the integer type variable pointed by `y`, e.g.,

```c
*y = 10;
```

These three steps can also be combined, e.g.,

```c
int * y = new int (10);
```
An example

Below shows an example on the relationship between pointers and arrays: one of the most important application of the pointer type.

```
#include <iostream.h>

typedef int* intPointer;

void main(){
    int Array[5]={0,1,2,3,4};
    intPointer p;

    p=Array;
    p=p+3;
    cout << *p << endl;
}
```

**Question:** What will be the output?

**Answer:** 3.
This Swap or that Swap?

In coding the sorting algorithms, we used a swap method to exchange the values of two variables. Will the following version work?

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

**Question:** What will the following program print out?

```c
void main(void){
    int x = 1, y = 2;

    swap(x, y);
    cout << x << y << endl;
}
```

**Question:** How could we fix it?

**Answer:** Use pointers.
Another swap?

Using the ideas of pointers, we can come up with another example of a swap function as follows:

```c
void swap(int* pa, int* pb){
    int temp = *pa;
    *pa = *pb;
    *pb = temp;
}
```

**Question:** What will the following program print out?

```c
void main(void){
    int x = 1, y = 2;

    swap(&x, &y);
    cout << x << y << endl;
}
```
Implement pointers and objects

Some other languages, such as Fortran, do not provide such types as pointers or references. But, we can still implement them with, e.g., the array data types.

We can represent a collection of objects containing same data fields by using an array for each data field, including the pointer fields implemented as array indices. Below gives a simple example, including allocating another node, and deallocating a node.
A single-array representation

The computer memory is essentially organized as a one dimensional array addressed by integers from 0 to $M - 1$, where $M$ is the memory size. In many programming languages, an object occupies a block of contiguous locations in the memory. A pointer to that object is simply the address of the first memory location allocated to that block. The location of the components of that object can be calculated as an offset to that pointer.

Thus, naturally, we can use a single array to represent an object. For example, the following figure shows how to represent linked lists with an array.

![Diagram showing single-array representation of linked lists]
Space allocation

To insert an object into a dynamic set represented by a doubly linked list, we have to allocate a pointer to a currently unused object first.

Thus, it makes sense to identify and organize those that are currently unused, also in a linked list. Whenever we need space for another object, we get such a piece from this unused list, and whenever we no longer need an object, we return its space to this unused list.

We now briefly discuss the issues involved with such allocating and freeing objects using a doubly linked list represented by multiple arrays.
Assume that the arrays have length $m$ and that at some point the dynamic set contains $n \leq m$ elements. Then, the remaining $m - n$ objects in the arrays are free, and will be organized as a free list. This list is implemented as a singly linked list, thus only using the next array to keep track of the unused objects. Below shows an example.
Implementation details

The free list is organized as a stack to save time. Thus, the allocation and the free procedures can be implemented as follows:

ALLOCATE-OBJECT()
1. if free=NIL
2. then error "out of space"
3. else x<-free
4. free<-next[x]
5. return x

FREE-OBJECT(x)
1. next[x]<-free
2. free<-x

**Homework:** Exercises 10.3-1(*), 10.3-3, and 10.3-4.
Representing rooted trees

We can also use a linked structure to represent rooted trees. A tree is more general than a list in the sense that for every element, it could have multiple successors. The simplest case is the *binary tree*, in which each element has two successors, a *left child* and a *right child*.

Hence, we can represent an element in a binary tree using an object with three fields: \( p \), \( \text{left} \), and \( \text{right} \) for its parent, left child and right child. In particular, if the \( p \) field of an element contains \( \text{NIL} \), it means that this element is the *root* of the tree.
An example

Below shows an example of representing a binary tree.
The more general case

When a tree has more than 2, but bounded number of, children, we can still use an extended version of the previous structure. Instead of left and right, we can use child1, child2,...childk, where k is the largest possible number of children any element could have.

Even for a tree with unbounded number of children, we still have the following way to represent such a tree by borrowing the binary tree representation: we use the left[x] field to point to the leftmost child of node x, and use the right[x] field to point to the sibling of x immediately to the right if there is one.
An example

Below shows an example of representing a general tree.

Homework: Exercises 10.4-1(*), 10.4-2(*), and 10.4-4. Problems 10-1(*).