CMSC 351 - Introduction to Algorithms
Spring 2012
Lecture 7

Instructor: MohammadTaghi Hajiaghayi
Scribe: Rajesh Chitnis

1 Introduction

In this lecture we give an introduction to Data Structures like arrays, linked lists, etc.

2 A Brief Introduction to Data Structures

Data structures are the building blocks of computer algorithms. Here we consider abstract data type (ADT), i.e., we do not specify the data types like integers, reals, characters, etc. It is more convenient and more general to design algorithms for these operations without specifying the data type of the items.

Modern object-oriented languages such as C++ or Java support a form of abstract data. When a class is used as a type, it is an abstract type that refers to hidden representation. An ADT is typically implemented as a class and each instance of the ADT is an object of that class.

3 Elementary Data Structures

3.1 Elements

It is a generic name for an unspecified data type. An element could be an integer, a set of integers, a file of text, or another data structure. We use this term whenever the discussion is independent of the type of data. For example, in sorting we only care about comparisons between element which can be integers, names (strings of characters), etc. Usually we assume we can compare elements for equality (and some times for "less than" relation) and we can copy them. Each of these operations is counted as one unit of time.

3.2 Arrays

An array is a row of elements of the same type. The size of an array is the number of elements in that array. The size of the array should be fixed and allocated in memory in advance. The elements can be bytes, integers, strings or more generally any elements. Arrays are very efficient and very common data structures and each element of it can be accessed in constant time. Arrays are good except they store elements of the same type and their size is fixed. We represent them as $a[12 \ldots n]$ and access element i as $a[i]$ for $1 \le i \le n$. We can also have a two-dimensional array $a[12 \ldots n, 12 \ldots m]$ and access a general element as $a[i, j]$.

3.3 Records

Records are similar to arrays, except that we do not assume all elements are of the same type. A records thus is a list of elements of different types. However like arrays the exact combination of types and thus the storage size is known (since the sizes of all the elements in the records are known). Each element in a record can be accessed in constant time. It is also possible to compute the location of each element in constant time. Let us now see an example of a Record:

begin

- Int 1: integer;
- Ar1: array[1 2 ... 20] of integers;
- Name1: array[1 2 ... 20] of characters;

end

Structures in C and classes in C++ (more general) can be used to implement records.

3.4 Linked Lists

There are many applications in which the number of elements is changing dynamically as the algorithm progresses (we can have large arrays to overcome this, but it is often not a good solution. Also due to consecutive representation of arrays doing some operations like insertion and deletion in the middle of an array is very inefficient). Linked lists are the simplest form of **dynamic** data structures. Here each element is represented separately and all elements are connected through the use of **pointers**. A pointer is simply a variable that holds as its value the address of another element. It exists in C, C++ but not explicitly in Java (although it can be simulated).

A linked list is a list of pairs, say in a record, each containing an element and a pointer, such that each pointer contains the address of the next pair. It is not possible to access each element directly in a linked list, instead one needs to scan it by following the addresses in the pointer. This is called as a linear scan. The end of the list will be denoted by nil or null, a pointer which points to nowhere. An example is given in Figure 1.

DRAWBACKS OF LINKED LISTS

1. Requires more space: one additional pointer per element (not a huge problem though).
2. Need a linear scan to find say the 30th element.

ADVANTAGES OF LINKED LISTS

1. With only 2 operations we can insert an element in the middle.
2. With only 1 change we can delete an element.

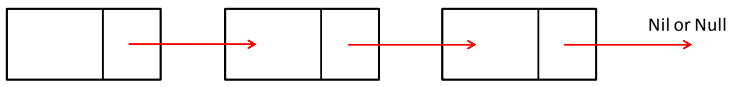

Figure 1: Example of a linked list

3.5 Stacks

Stacks are dynamic data structures in which we delete (**pop**) an element which was most recently inserted (**push**). A stack implements a last-in-first-out (LIFO) policy. We have also an operation **top** which gives the element at the top of the stack. We can implement it with both arrays (if we know the maximum size) or linked lists. We now show how to implement the Push, Top and Pop functions using arrays. As an exercise, you can implement them with linked lists. Each of Push, Pop and Top operations takes $O(1)$ time.

Algorithm 1 Pop(s)

```
1: if t=0 then
2:     return error "stack underflow"
3: else
4:     $t \leftarrow t - 1$
5:     return $s[t + 1]$
6: end if
```

Algorithm 2 Push(s, x)

```
1: $t \leftarrow t + 1$
2: $s[t] \leftarrow x$
```

Algorithm 3 Top(s)

```
1: if t=0 then
2:    return error “stack underflow”
3: else
4:    return $s[t]$
5: end if
```

3.6 Queues

A queue is a dynamic data structure in which the element deleted (**dequeue**) is always the one that has been inserted (**enqueue**) for the longest time. A queue implements a first-in-first-out (FIFO) policy like in the registrar’s office. The queue has a head and a tail. We insert at the tail and delete from the head. It can be implemented by array or linked lists (like stacks). We now show how to implement the Enqueue and Dequeue functions using arrays as if the queue is circular. As an exercise, you can implement them with linked lists. Each of Enqueue and Dequeue operations takes $O(1)$ time.

Algorithm 4 Enqueue(Q, x)

```
1: $Q[tail] \leftarrow x$
2: if tail = lengthQueue then
3:    tail $\leftarrow 1$
4: else
5:    tail $\leftarrow$ tail $+1$
6: end if
```

References

[1] Udi Manber, *Introduction to Algorithms - A Creative Approach*

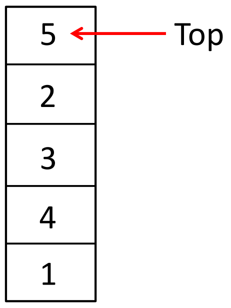

Figure 2: Example of a stack

Algorithm 5 Dequeue(Q, x)

```
1: $x \leftarrow Q[head]$
2: if head = lengthQueue then
3:     head $\leftarrow 1$
4: else
5:     head $\leftarrow$ head $+1$
6: end if
7: return $x$
```

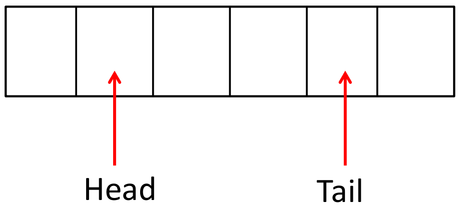

Figure 3: Example of a queue